НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Кафедра обчислювальної техніки

«На правах рукопису»
УДК __004.056.53___

До захисту допущено:

Завідувач кафедри

_____ *Сергій СТІРЕНКО*
«___»_____2021 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Комп'ютерні системи та мережі»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Метод та засоби побудови хеш-перетворення з програмованими колізіями для систем строгої ідентифікації віддалених користувачів»

Виконав:
студент VI курсу, групи ІВ-91мн
Бояршин Ігор Іванович _____

Керівник:
доцент, к.т.н., доцент,
Марковський Олександр Петрович _____

Консультант з нормоконтролю:
професор, д.т.н., професор,
Кулаков Юрій Олексійович _____

Рецензент:
декан ФПМ, д.т.н, професор,
Дичка Іван Андрійович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет (інститут) *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-науковою програмою
Спеціальність *123. Комп'ютерна інженерія*
(код і назва)

Спеціалізація *123. Комп'ютерні системи та мережі*
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Стіренко С.Г.
(підпис) (ініціали, прізвище)
« » 20***21*** р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Бояршину Ігорю Івановичу
(прізвище, ім'я, по батькові)

1. Тема дисертації Метод та засоби побудови хеш-перетворення з програмованими колізіями для систем строгої ідентифікації віддалених користувачів

Науковий керівник дисертації доц.,к.т.н., доц. Марковський О.П.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «12» березня *2021* р. № 809-с

2. Строк подання студентом дисертації *26 квітня 2021 р*

3. Об'єкт дослідження *процеси ідентифікації віддалених користувачів що базуються на використанні незворотних булевих перетворень для реалізації криптографічно-строгої концепції нульових знань.*

4. Предмет дослідження *методи побудови булевих незворотних і неоднозначних булевих перетворень, стійкий до лінійного і диференційного криптоаналізу.*

5. Перелік завдань, які потрібно розробити: *огляд існуючих методів та засобів ідентифікації, визначення можливостей підвищення ефективності та шляхів*

реалізації цих можливостей, розробка методу строгої ідентифікації віддалених користувачів на основі застосування незворотних неоднозначних булевих функціональних перетворень, створення програмних засобів автоматизації проектування булевих функціональних перетворень на основі концепції "нульових знань", теоретична та експериментальна оцінка ефективності використання неоднозначних незворотних булевих перетворень для прискорення строгої ідентифікації користувачів.

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Кулаков Ю.А., професор		

7. Дата видачі завдання 26.11.2020

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1.	*Затвердження теми роботи*	*10.12.2020-15.12.2020*	
2.	*Вивчення та аналіз завдання*	*15.12.2020-31.01.2021*	
3.	*Розробка архітектури та загальної структури системи*	*01.02.2021-10.02.2021*	
4.	*Розробка структур окремих підсистем*	*10.02.2021-20.02.2021*	
5.	*Програмна реалізація системи*	*20.02.2021-10.04.2021*	
6.	*Оформлення пояснювальної записки*	*10.04.2021-25.04.2021*	
7.	*Передзахист*	*26.04.2021*	
8.	*Захист*	*17.05.2021*	

Студент _____ І.І. Бояршин
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ О.П. Марковський
(підпис) (ініціали, прізвище)

РЕФЕРАТ

на магістерську дисертацію

виконану на тему: Метод та засоби побудови хеш-перетворення з програмованими колізіями для систем строгої ідентифікації віддалених користувачів

студентом: Бояршиним Ігорем Івановичем

Робота складається зі вступу та чотирьох розділів. Загальний обсяг роботи: 94 аркуші основного тексту, 5 ілюстрації, 9 таблиць, додатки. При підготовці використовувалася література з 47 різних джерел.

Актуальність. Динамічний розвиток мережевих технологій стимулює кількісний та якісний розвиток розподілених систем. В рамках таких систем широкому колу користувачів стали доступні значні за обсягом інформаційні ресурси, ресурси пам'яті та обчислювальні потужності. Ідентифікація користувачів таких систем є важливою ланкою в організації контролю за доступом до цих ресурсів. Розширення використання ресурсів розподілених систем, в тому числі за рахунок застосувань, інформаційні ресурси яких є конфіденційними, вимагає підвищення надійності контролю доступу до цих ресурсів. Разом з тим, випереджаючими темпами в порівнянні з продуктивністю комп'ютерних систем зростає кількість їх користувачів. Це вимагає підвищення продуктивності контролю за доступом до ресурсів розподілених систем. Таким чином, актуальною є проблема підвищення ефективності ідентифікації користувачів розподілених систем, як в плані зменшення ризиків неконтрольованого доступу до їх ресурсів, так і в плані прискорення процесу ідентифікації.

Мета і завдання дослідження. Метою магістерської роботи є підвищення ефективності ідентифікації користувачів у багатокористувацьких системах.

Для досягнення мети дослідження поставлено і вирішено такі задачі :

1. Аналіз сучасного стану забезпечення інформаційної безпеки та розділення

прав доступу систем колективного доступу, обґрунтування критеріїв ефективності ідентифікації віддалених користувачів; огляд, з позицій цих критеріїв існуючих методів та засобів ідентифікації, визначення можливостей підвищення ефективності та шляхів реалізації цих можливостей.

2. Вдосконалення методу строгої ідентифікації віддалених користувачів на основі застосування незворотних неоднозначних булевих функціональних перетворень. Вибір структури обчислення таких перетворень та розробка методу їх синтезу, який забезпечує їх стійкість до лінійного та диференційного криптоаналізу.

3. Створення програмних засобів автоматизації проектування булевих функціональних перетворень спеціальних класів для прискореної ідентифікації віддалених користувачів на основі теоретичної концепції “нульових знань”.

4. Теоретична та експериментальна оцінка ефективності використання неоднозначних незворотних булевих перетворень для прискорення строгої ідентифікації користувачів.

Об’єкт дослідження – процеси ідентифікації віддалених користувачів що базуються на використанні незворотних булевих перетворень для реалізації криптографічно-строгої концепції нульових знань.

Предмет дослідження – методи побудови булевих незворотних і неоднозначних булевих перетворень, стійкий до лінійного і диференційного криптоаналізу.

Методи досліджень базуються на базових засадах теорії булевих функцій, теорії рекурсивних функцій, теорії ймовірності, а також на основних положеннях статистичного та імітаційного моделювання.

Наукова новизна одержаних результатів роботи полягає у наступному:

Вдосконалено метод криптографічно-строгої ідентифікації віддалених користувачів на основі незворотних булевих функціональних перетворень, який відрізняється від відомих використанням табличних підстановок, кількість яких

зменшується в кожному наступному шарі перетворення, що дозволяє збільшити кількість сеансових ключів ідентифікації.

Вдосконалено метод криптографічно-строгої ідентифікації віддалених користувачів на основі незворотних булевих функціональних перетворень, який відрізняється тим, що булеві перетворення набувають властивостей лавинного ефекту, що забезпечує їх стійкість до диференційного криптоаналізу.

Практична значимість отриманих результатів визначається тим, що хеш-перетворення, які використовуються для реалізації криптографічно-строгої ідентифікації є більш стійкими до спроб їх порушення методом диференційного криптоаналізу. Збільшена кількість сеансових паролей, яку забезпечує розроблений метод дозволяє збільшити число циклів ідентифікації до пере налаштування системи.

Особистий внесок здобувача полягає в теоретичному обґрунтуванні одержаних результатів, їх експериментальній перевірці та дослідженні, а також у створенні програмних продуктів для практичного використання одержаних результатів.

Апробація результатів дисертації

Основні результати дисертації доповідались та обговорювались на 2-х міжнародних науково-технічних конференціях:

1. Міжнародна наукова конференція “Security, Fault Tolerance, Intelligence: ICSFTI2019”. м.Київ, 14-15 травня 2019 р.

2. Міжнародна наукова конференція “Security, Fault Tolerance, Intelligence: ICSFTI2020”. м.Київ, 13 травня – 15 липня 2020 р.

Публікації

Основні положення магістерської дисертації опубліковані в 3 наукових працях, серед яких дві – матеріали наукових конференцій та одна – стаття у фаховому журналі.

1. Boyarshin I. Method of hash transformations construction for strict user

identification / Igor Boyarshin, Oleksandr Markovskyi // International Conference ICSFTI2019 (Kyiv, May 14−15, 2019). Kyiv, 2019. – P. 41-46.

2. Rusanova O. Energy-aware task scheduling algorithm for mobile computing / Olga Rusanova, Igor Boyarshin, Anna Doroshenko // International Conference ICSFTI2020 (Kyiv, May 13, June 15, 20120). Kyiv, 2020. – P. 107-113.

3. Boyarshin I. Request balancing method for increasing their processing efficiency with information duplication in a distributed data storage system / I. Boyarshin, A. Doroshenko, P. Rehida // Technical sciences and technologies. – 2021. – № 2 (26).

Ключові слова

Криптографічно-строга ідентифікація, хеш-перетворення, булеві функціональні перетворювачі, концепція «нульових знань», багатокористувацька система.

ЗМІСТ

ВСТУП

Останні роки все більше зростають темпи інтеграції комп’ютерних технологій в усі сфери людської діяльності. По мірі вдосконалення обладнання та програмного забезпечення, що використовується, з’являються нові шляхи використання технологій, що дозволяє спрощувати або навіть робить можливим речі, які раніше не можна біло втілити у життя. Так, значний стрибок зробили хмарні технології, що дають змогу надати доступ будь-кому до великих об’ємів ресурсів, зокрема до обчислювальних систем та систем зберігання даних.

Одним із основних рушійних чинників прогресу є комерційний аспект, а тому більшість сервісів, які зараз пропонують свої послуги з надання доступу до віддалених ресурсів, є комерційними у своїй основі. Саме тому головним питанням тепер є забезпечення достатнього рівня захисту даних, а також створення можливості доступу до ресурсів багатьом користувачам одночасно, що призводить до необхідності введення системи контролю за правами доступу користувачів. Одним із основних аспектів при цьому є ідентифікація користувачів у системі, а отже важливим є питання забезпечення швидкісного та безвідмовного рівня виконання операції ідентифікації у системах багатьох користувачів.

У 21-му століття все більш популярними стають системи, що надають доступ до інформації багатьох користувачів в один момент часу. Через це виникає необхідність протистояння несанкціонованому доступу до сервісів системи та даних користувачів. Можна виділити дві причини розповсюдження несанкціонованого доступу. По-перше, це підвищення апаратних можливостей сучасних комп’ютерних систем, а саме їх швидкодії. По-друге, зростають відповідні “винагороди” у разі вдалого незаконного доступу до даних окремих користувачів або системи загалом. Не можна забувати і про той фактор, що на сьогоднішній день зловмисник не є обмеженим власними обчислювальними ресурсами: йому так само відкритий доступ до хмарних обчислень, які по своїм

можливостям як правило перевищують можливості звичайних обчислювальних систем.

У питанні ідентифікації користувачів у системах віддаленого доступу найкраще себе зарекомендувала криптографічно-строга ідентифікація. У її основі покладена концепція існування лише одного власника секретних даних у будь-яких момент часу, а також передбачається, що на кожному сеансі ідентифікації буде використано черговий (новий) пароль користувача. Набір цих правил отримав назву концепції “нульових знань”. На практиці успішне застосування цієї концепції обмежено лише обчислювальними ресурсами, які використовуються для її реалізації. Це пояснюється, по-перше, великими об’ємами дискового простору та оперативної пам’яті, що потрібні для її виконання, і по-друге, значними затратами обчислювальної потужності. Другий фактор виникає через те, що системи строгої ідентифікації, які існують на сьогоднішній день, засновані на модулярному експоненціюванні чисел великої розрядності, розрядність яких може досягати 2048 або навіть 4096 біт. Важливою особливістю таких обчислень є той факт, що зі збільшенням розрядності чисел, над якими виконуються операції, відповідні часові затрати збільшуються експоненційно. З ростом швидкісних можливостей комп’ютерів виникає необхідність використовувати все більшу і більшу розрядність чисел для забезпечення принаймні того самого рівня захисту. А отже, оскільки темпи зростання обчислювальних можливостей апаратної бази є меншими за темпи зростання обчислень, що потрібно виконати, то ці обчислення, закономірно, потребують все більше часу.

Найбільш гостро питання кількості необхідних обчислень, що потрібно виконати, постає при використанні мобільних пристроїв, що працюють на малопотужних мікроконтролерах для подовження часу їх автономної роботи. Через це питання ефективних алгоритмів та способів реалізації належного рівня захисту стає ще більш актуальним.

Отже, питання пришвидшення ідентифікації користувачів у системах надання розподіленого віддаленого доступу до сервісів та інформації завдяки запровадженню концепції “нульових знань” є найбільш актуальної на сьогоднішній день при наявному рівні розвитку інформаційних технологій.

РОЗДІЛ 1

АНАЛІЗ СУЧАСНИХ ВИМОГ ДО ЗАСОБІВ ІДЕНТИФІКАЦІЇ ТА КРИТИЧНИЙ ОГЛЯД ІСНУЮЧИХ ЗАСОБІВ

1.1. Аналіз сучасних систем інтегрованого доступу та обґрунтування вимог до ідентифікації їх користувачів

Серед базових задач, які вирішує система захисту інформації, входить ідентифікація користувачів [1]. Вона відповідає за унеможливлення несанкціонованого доступу до сервісів системи багатьох користувачів загалом та даних окремих користувачів.

Будь-яких сценарій взаємодії віддаленого користувача з багатокористувацькою системою передбачає реєстрування користувача у базі даних системи [2]. Ця процедура виконується один раз, після чого в подальшому користувач буде проходити лише ідентифікацію в системі. При цьому під ідентифікацією мається на увазі авторизація користувача у системі, тобто визначення та підтвердження його особистості. Зазвичай кожному користувачу призначається унікальний ідентифікатор, тобто встановлена взаємо-однозначна відповідність між множиною ідентифікаторів та множиною зареєстрованих користувачів.

З ростом масштабів обчислювальних систем все більш розповсюдженими стають розподілені системи. Це в свою чергу означає, що необхідно мати захист як від локальних загроз, так і від загроз, що надходять по мережі. Серед найбільш популярних атак на системи багатьох користувачів [3] можна віднести наступні:

1. Доступ зловмисника безпосередньо до обчислювальної системи, що передбачає його фізичну присутність;

2. Злом чи обман системи захисту зловмисником, що дає йому незаконний доступ до головної системи;

3. Так звана “підміна користувача”, тобто ситуація, коли зловмиснику вдається видати себе за користувача, яким від насправді не є. Зазвичай вибирається користувач з підвищеними привілеями доступу;

4. Так звана “підміна серверу”, тобто ситуація, коли зловмиснику вдається видати свою систему (сервер) за справжню у каналі спілкування користувача з сервером. Це призводить до отримання доступу до даних користувача на справжньому сервері;

5. Отримання зловмисником незаконного доступу до даних користувача. Це призводить до маніпулювання користувачем, коли його спричиняють до певних дій під загрозою втрати його даних;

6. Незаконне втручання зловмисника до даних користувача з метою їх зміни.

Більшість існуючих систем захисту інформації будуються на основі такого механізму ідентифікації, що передбачає підтвердження володіння користувачем певної секретної інформації. У такій моделі взаємодії користувач і система позначаються як A і B, відповідно [3]. При цьому система B зберігає дані користувачів з множини $\Omega=\{A_1, A_2, \ldots, A_n\}$, де кожен з користувачів з цієї множини має право доступу до своїх даних, що зберігаються в системі B.

Найбільш популярним способом оцінки якості системи ідентифікації є перевірка їх стійкості до атак, що передбачають спроби користувачів з множини $Q \not\in \Omega$ отримати доступ до системи шляхом успішного проходження ідентифікації. При цьому замірюється час та об’єм обчислювальних ресурсів, які використовуються при проходженні ідентифікації.

Вважається, що система ідентифікації віддалених користувачів є тим більш якісною, чим більше ресурсів необхідно витратити на незаконний злом та подальший доступ до системи шляхом успішного проходження ідентифікації, та

чим менше часу витрачається на штатний процес ідентифікації. Серед ресурсів, за допомогою яких вимірюється якість системи ідентифікації, виділяють наступні [4]:

- час виконання алгоритму;
- об’єм використаної оперативної чи дискової пам’яті;
- об’єм даних, який був пропущений через мережеву лінію.

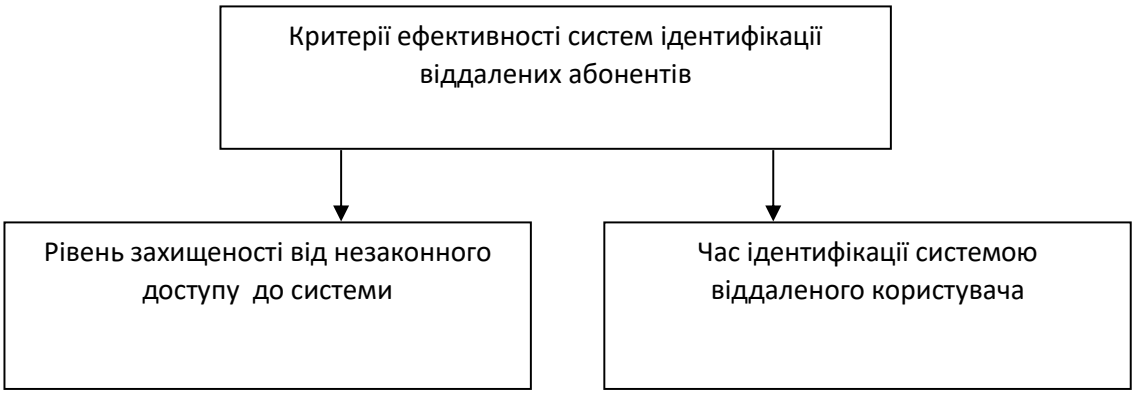

Рис.1.1. Фактори, що використовуються при оцінці якості системи ідентифікації

Будь-яка система ідентифікації віддаленого користувача будується на основі того, що користувач має унікальний доступ до певної секретної інформації, а також механізму підтвердження її коректності. Як наслідок, можна виділити такі дві стадії алгоритму ідентифікації користувача у системі:

1. Стадія реєстрації користувача, коли користувачу надається унікальний доступ до певної секретних даних S_A, що їх він в подальшому використовуватиме на стадії другій ідентифікації.

2. Стадія ідентифікації користувача, коли від користувача вимагається підтвердити володіння ним секретних даних S_A, які він отримав на першій стадії реєстрації.

Для інтегрованих систем, що займаються обробкою даних, базовою та головною вимогою в контексті ідентифікації користувачів в системі є

унеможливлення для зловмисника (користувача, дії якого не я санкціонованими) отримання доступу до системи. Далі будуть розглянуті основні способи, які зазвичай використовуються для проникнення в інтегровані системи з частковим доступом. Це дасть змогу сформулювати точніші завдання, що постають перед подібними системами ідентифікації. Вважається, що інтегрована система може опрацювати n віддалених користувачів, що входять до певної множини Ω. При цьому кожен j-й користувач, де j=1,2..,n, володіє певними унікальними секретними даними K_j і певними даними для власної ідентифікації I_j, а також власними даними користувача D_j. Користувач має право отримати доступ до власник даних D_j тоді і тільки тоді, коли зміг продемонструвати та підтвердити володіння секретними даними K_j, що належать ідентифікатору I_j.

Таким чином, злом подібної системи буде виглядати наступним чином (зводитися до таких випадків). По-перше, користувач, що не є законним користувачем, тобто не входить до раніше описаної множини Ω, зможе отримати доступ до секретних даних інтегрованої системи. По-друге, звичайний h-й законний користувач з множини законних користувачів $A_h \in \Omega$ отримає доступ до секретних даних іншого законного користувача B_j (причому $j \neq h$).

Також можна виділити такі способи отримання доступу незаконного користувача, за умови що він має доступ до мережі:

- Повторення спроби проходження ідентифікації багато разів за допомогою ідентифікатора I_h. При цьому найбільше значення необхідних спроб завжди існує і є конкретним в кожному випадку, проте може сильно варіюватися і зазвичай залежить від багатьох факторів;
- Так званий “сніффінг”, тобто ситуація, коли прослуховується лінія зв’язку існуючого законного користувача з системою на предмет наявності та отримання ідентифікаційної інформації цього користувача;

- Різновид “сніффінгу”, коли необхідно перехопити лише частину секретної інформації в момент проходження законним користувачем ідентифікації з метою змінення частини цієї інформації та використання її у майбутньому;
- Злом самої системи з метою отримання даних користувачів, які знаходяться там вже тривалий час.

З кожним з перерахованих варіантів можна проасоціювати певну величину ресурсів (часових або обчислювальних), що їх зловмисник має витратити на втілення конкретного варіанту в життя. І для кожного з них має бути готовий певний механізм протидії йому з боку системи, аби не дати зловмиснику незаконного доступу до даних системи.

Наприклад, одним із базових механізмів протидії незаконним втручанням до системи є заборона використання одного й того самого паролю користувача декілька разів, а також унеможливлення отримання доступу до системи за допомогою уже використаного паролю. Це дозволяє протидіяти сценарію злому, коли зловмисник зміг отримати доступ до паролю законного користувача, підслухавши його сеанс ідентифікації з системою. Для реалізації такого механізму протидії найбільш простим методом є використання так званих “часових штампів” [5], таким чином системі потрібно перевірити не тільки пароль користувача, але це й те, що він є дійсним з точки зору часу. Іншим методом є втілення протоколу, що передбачає багаторазове спілкування користувача з системою та навпаки, де на кожному кроці пересилаються та підтверджуються різні дані, частина з яких генерується за допомогою генератора випадкових чисел [6]. Додаткове підвищення надійності системи при використанні другого методу досягається також тим, що кількість успішних етапів, що потрібно пройти зловмиснику, збільшується, що в свою чергу зменшує шанс його успіху.

Підсумувавши сказане вище, до системи ідентифікації висуваються такі вимоги, які здатні протидіяти описаним методам отримання доступу зловмисником до системи:

- На кожному сеансі проходження ідентифікації користувачем у системі ним мають використовуватися інші, нові дані для ідентифікації, адже в іншому випадку підвищується ймовірність отримання та повторного використання цих даних зловмисником;
- При проходженні ідентифікації у системи користувач має використовувати лише частину секретних даних, що зберігаються лише у нього та не передаються по мережі до системи. Окрім процесу ідентифікації, ці дані мають точно вказувати, до яких саме даних та сервісів користувач отримає доступ;
- У разі незаконного проникнення зловмисника до системи, вона не має містити жодних даних, отримавши які, зловмисник зміг би дізнатися дані, необхідні для успішного проходження ідентифікації користувачем у системі;
- Використання особливих методів та засобів захисту даних при виконанні перевірки коректності та дійсності ідентифікуючої інформації користувача;
- Оскільки більшість систем, де проходить ідентифікація користувача, обробляють безліч запитів користувачів одночасно, то важливим аспектом є швидкість проходження ідентифікації одним користувачем, аби, з одного боку, забезпечувати належний рівень захисту, а з іншого, дозволяти системі мати високу інтенсивність обслуговування користувачів.

При цьому необхідно зауважити наступні особливості обчислювальних можливостей користувача на системи. Оскільки система зазвичай є розподіленою та високошвидкісною, у той час як користувачу зазвичай доступні лише невеликі обчислювальні ресурси, то з’являється можливість виконати більш складні обчислення з боку системи під час проходження ідентифікації користувачем, в той

час як обчислювальні затрати користувача мають бути зменшені по мірі можливості.

1.2. Огляд теоретичних засад ідентифікації. Слабка та строга ідентифікація

На сьогоднішній день найбільш популярним засобом проведення ідентифікації є використання пароля користувача у якості його ідентифікатора. Головним недоліком такого підходу є людський фактор, коли користувач використовує пароль, що не надає належний рівень захисту через те що є простим або недостатньо великим, а значить вразливим для атак зловмисників. Через це механізм ідентифікації, що засновані лише на використанні паролю, не є достатньо надійними. Одним із варіантів обходу цієї проблеми є використання спеціальних алгоритмів для генерації псевдовипадкових паролів, що складаються з випадкових символів та мають достатньо велику довжину [8]. З одного боку, це набагато ускладнює злам системи, але з іншого, покладає на користувача необхідність запам’ятовувати цей складний пароль. Ситуація ускладнюється ще більше, коли користувачу необхідно мати справу з такими комплексними паролями у декількох сервісах одночасно.

До подібних систем висувається перелік додаткових рівнів безпеки, яким вони мають відповідати, аби бути надійними. До них можна віднести як необхідність автономної роботи у разі втрати живлення, так і достатній рівень захисту від зловмисників у разі незаконного впливу на них.

Одним із основних питань, які мають бути вирішені для задачі проведення ідентифікації в системі, є знаходження компромісу між швидкодією системи на рівні захисту, який вона здатна забезпечити. Для надання належного рівня сервісу система має мати достатню швидкодію своєї системи ідентифікації. Це означає зменшення кількості обчислювальних завдань, що потрібно виконати на кожному

сеансі ідентифікації. Але це в свою чергу не має впливати на рівень захисту, що надається.

Останнім часом набуває популярність такий механізм ідентифікації, що передбачає використання користувачем комбінації двох складових, а саме логіну та паролю [2]. У такому сценарії кожному користувачу відповідає унікальний логін, який використовується при проведенні ідентифікації. Отримавши логін, система виконує його пошук у базі даних та, у разі успішного знаходження, порівнює введений користувачем пароль із тим, що зберігається у базі. Нарешті, доступ до системи надається користувачу тоді і тільки тоді, коли і логін і пароль виявилися правильними. Варто зауважити, що паролі не мають обов’язково бути унікальними для всіх користувачів, воли лише мають відповідати тим, що зберігаються для відповідного логіну у системі.

Не дивлячись на простоту такого відходу, цей метод має також певні недоліки:

– Не виключена можливість отримання зловмисником доступу до бази даних системи, де зберігаються логіни та паролі користувачів;

– Одним із можливих сценаріїв обходу системи захисту у такому випадку є видання зловмисником себе за адміністратора системи з метою примушення користувача видати свій пароль, або створити новий за вказівкою зловмисника. Або ж навпаки, зловмисник видає себе за звичайного користувача і намагається примусити систему змінити пароль для нього на новий;

– Оскільки паролі вибираються користувачем з метою легкого запам’ятовування, то нерідко вони засновані на якихось особистих даних. Також одним із сценаріїв дій у разі втрати паролю є підтвердження у системі якихось особистих даних користувача. Отримавши доступ до таких особистих даних, шанси зловмисника на доступ до системи від імені цього користувача підвищуються;

– Як завжди, залишається варіант “сніфінгу” мережі зловмисником для вилучення даних користувача, що можуть бути використані для ідентифікації.

Одним із можливих покращень є використання системи автоматичного управління паролями. Така система передбачає, що користувач має створити та запам’ятати єдиний головний пароль, що має бути достатньо складним. Сама система ж відповідає за створення, зберігання та використання унікальних складних паролів для усіх сервісів, куди користувач доступається. Таким чином, перехват будь-якого з паролів не дає зловмиснику доступу до інших сервісів користувача, адже паролі є унікальними. Також, оскільки паролі є складними та генерується випадковим чином, злам будь-якого з них є надто складною задачею.

Як уже описувалося раніше, у будь-якому випадку існує встановлена взаємно-однозначна відповідність між множиною унікальних ідентифікаторів, що використовуються протягом ідентифікації у системі користувача, та множиною користувачів.

При використанні системи автоматичної генерації та зберігання паролів, у технічній літературі вона зазвичай називається центром генерації та розподілу ключів — ЦГРК [11], або просто позначається через S. Схему з використанням системи автоматичної генерації та зберігання паролів зображено на рис. 1.2, де показано її взаємодію з користувачем та системою, до якої користувач доступається. Такий механізм передбачає виконання трьох основних пунктів, зокрема: генерація паролів, реєстрація паролів та комунікація із системою. Перший етап генерації передбачає створення системою S множини нових відкриваючих та закриваючих криптографічних ключів. Другий етап реєстрації передбачає генерацію для кожного користувача нового ідентифікатора та паролю (секретної інформації, що використовуватиметься при ідентифікації). Третій етап комунікації передбачає виконання самого зв’язку між клієнтом та сервером з використанням ідентифікуючої інформації, згенерованої раніше. При цьому відбувається

створення нового сеансового ключа. Всі дії з сеансовим ключем проходять через ЦГРК.

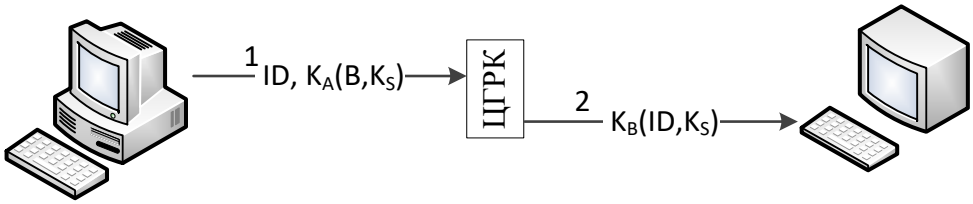

Рис. 1.2. Ідентифікація з використанням ЦГРК

Описаний вище механізм базується на наступній послідовності дій. Користувач має намір відкрити канал з системою за допомогою ключа сеансу K_S, про що він сповіщає ЦГРК. Для шифрування даного повідомлення користувача використовується секретний ключ K_A, особливістю якого є те що про нього знають лише сам користувач і ЦГРК [19]. Розшифрувавши отримане від користувача повідомлення, ЦГРК вилучає з нього ідентифікуючу інформацію для системи, а також сеансовий ключ. Далі за допомогою сеансового ключа та користувацького ідентифікатора будується нове повідомлення у ЦГРК, після чого це повідомлення надсилається до цільової системи. Саме ж повідомлення шифрується ключем K_B, про який знають лише сама цільова система та ЦГРК. Наступним кроком є отримання повідомлення цільовою системою та розшифрування його. Зробивши це, система розуміє, що користувач бажає створити сеанс з використанням ключа K_S. Таким чином, увесь механізм ідентифікації відбувся автоматично, адже ЦГРК впевнений у достовірності користувача, бо саме від нього надійшло перше повідомлення, і тільки він мав засоби зашифрувати своє повідомлення ключем K_A. Аналогічно, цільова система впевнена, що друге повідомлення надійшло саме від

ЦГРК, бо лише він має доступ до секретного ключа K_B, не враховуючи саму систему.

З одного боку, використання такого механізму ідентифікації є досить надійним через свої переваги, проте, він є вразливим до так званої “атаки повторного відтворення” [1].

Альтернативним способом проведення ідентифікації є її модифікація, коли використовується спільний секретний ключ. Такий механізм ідентифікації використовується, наприклад, у системах, що використовують протокол оклик-відгук [12]. Він заснований на ідеї, що знайшла популярність у багатьох механізмах ідентифікації, коли сторона А генерує та надсилає стороні Б псевдовипадкове число, в той час як сторона Б опрацьовує це число особливим алгоритмом та повертає стороні А результат. Таким чином, процес ідентифікації користувача складається з етапів, зображених на рис. 1.3.

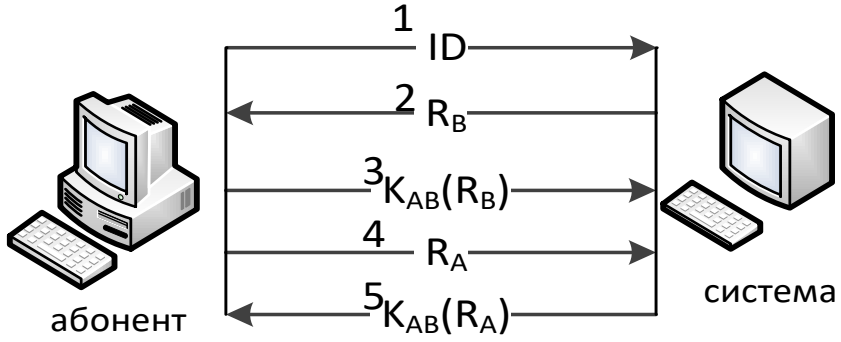

Рис. 1.3. Ідентифікація, що виконується на основі оклику-відгуку

Описаний вище механізм ідентифікації складається з таких етапів:

1. Спочатку користувач надсилає свій унікальний ідентифікатор до системи;
2. Система у відповідь користувачу генерує псевдовипадкове число великої довжини та надсилає його назад;

3. Вважається, що існує спільний для користувача та системи секретний ключ, яким користувач шифрує своє повідомлення, після чого надсилає його системі;

4. Після отримання повідомлення система за допомогою спільного ключа перевіряє, що повідомлення було зашифровано саме цим ключем;

5. Для перевірки того, що користувач комунікує зі справжньою системою, він надсилає до системи оклик, у відповідь на який система відправить йому відгук.

Існує ряд стандартних криптографічних алгоритмів, що можуть бути використані для системи ідентифікації користувачів. Їх особливістю є, по-перше, достатньо високий рівень криптостійкості, а по-друге, завдяки спеціальним апаратним засобам — можливість пришвидшити їх виконання на декілька порядків.

Найбільшу популярність [23] знайшли криптографічні алгоритми, що працюють за принципом відкритого ключа, адже саме вони якнайкраще відповідають принципу єдиного власника секретних даних [13].

Більшість алгоритмів, які знайшли застосування на сьогоднішній день, працюють на базі комплексних перетворень з розділу математики теорії чисел. Існуючі алгоритми можна розділити на симетричні та несиметричні.

Несиметричні алгоритми показують значно кращі результати при використанні їх для захисту інформації, при цьому один з двох ключів залишають відкритим, а інший зберігають в секреті.

Цей розділ криптографії (з використанням відкритого ключа) почав стрімко розвиватися на початку 1980-х років [18] і з тих пір знайшов реалізацію у багатьох механізмах захисту інформації.

Найбільш відомими та такими, що широко застосовуються, є наступні алгоритми: RSA, Мерклі-Хеллмана та ECC.

Іншим напрямком розвитку механізмів ідентифікації є такий, що використовує для її проведення певні фізичні предмети. Найбільш простим

варіантом такого механізму є використання зв’язки паролю або пін-коду користувача, що слугує його унікальним ідентифікатором, та фізичного предмету, як-от usb-накопичувача чи картки з магнітною доріжкою, або ж NFC-мітки. Для того аби запобігти отримання доступу до інформації зловмисником, який отримав доступ до цього спеціального фізичного предмету, він використовується у парі з паролем.

Смарт-картки та NFC-мітки можна розділити на дві категорії в залежності від роботи, яку вони виконують у механізмі ідентифікації.

Перший тип містить у собі незначний об’єм пам’яті, що зазвичай не перевищує декількох сотень байт, та призначений для зберігання базових даних, необхідних для проведення ідентифікації. Як різновид цього способу, серед цих даних може зберігатися у зашифрованому вигляді і сам пароль користувача, аби система, яка зчитує картку чи мітку, мала можливість провести перевірку користувача без наявності зв’язку з головною системою [19]. У базовому варіанті пароль користувача шифрується ключем, що зберігається лише у головній віддаленій системі. Не дивлячись на переваги такої системи, вона має і деякі недоліки. Зокрема, оскільки собівартість обладнання для подібної системи є досить низьким і воно є розповсюдженим, то збільшується ризик її зламу.

Другий тип накопичувачів містить окрім пам’яті в декілька тисяч байт ще й мікросхему, здатну проводити більш складні обчислення [18]. Подібні накопичувачі містять певну контрольну суму від даних користувача, а отже їм не загрожує втрата живлення від час роботи [15].

Наступною ітерацією цієї технології є використання смарт-накопичувачів з більш складним механізмом захисту даних [20]. Вони містять 8-бітний процесор з невеликою тактовою частотою та невелику пам’ять, що не перевищує декілька тисяч байт. Окрім обчислювальних можливостей, така карта здатна проводити повноцінний канал зв’язку з пристроєм зчитування. Подібна картка захищена від

зовнішнього впливу, а значить може бути використана для ідентифікації абонента в системі до створення повноцінного каналу зв’язку [21].

Усі розглянуть у цьому розділі алгоритми та механізми ідентифікації користувача мають один спільний недолік, а саме те, що по мірі покращення технологій вони рано чи пізно будуть зламані.

1.3. Огляд існуючих методів ідентифікації на основі концепції нульових знань

Можна виділити 2 класи схем ідентифікації: “слабка” ідентифікація, що була описана вище, яка базується на використанні паролю у якості основного механізму ідентифікації користувача; та “строга” ідентифікація, що бере за основу механізм концепції “нульових знань”.

При використанні першого типу ідентифікації, тобто ідентифікації з використанням звичайних паролів, не виконується головний принцип одного власника секретної інформації. Саме через це вона називається “слабкою”.

З іншого боку, “строга” ідентифікація виконується в рамках теорії “нульових знань”. Вона базується на тому принципі, що лише користувач володіє певною секретною інформацією, і має довести неявним чином цей факт. Сама система не володіє цими даними, проте має механізм перевірки їх правильності для підтвердження наявності секретних даних.

Сильною стороною такого підходу, а саме відсутність зберігання секретних даних користувача на стороні системи, автоматично унеможливлює отримання будь-якої корисної інформації зловмисником у разі отримання незаконного доступу до системи. Звідси і назва “нульові знання”. Другим базовим пунктом цієї концепції є використання користувачем на кожному сеансі ідентифікації нового унікального паролю, який це не був використаний раніше.

На основі усього цього можна зробити висновок, що саме “строга” ідентифікація на основі концепції “нульових знань” найкраще відповідає вимогам, що були сформовані вище до системі ідентифікації віддалених користувачів. Таких механізм ідентифікації засновується на використанні незворотних криптографічних перетворень. Під перетвореннями, що є незворотними, розуміють такі перетворення, які досить легко виконується (обраховуються) в одну сторону, проте знаходження оберненого перетворення в іншу сторону вважається неможливим через занадто великі часові та обчислювальні затрати на його знаходження.

Серед найбільш популярних на практиці механізмів зі “строгими” перетвореннями можна виділити FESIS, Guillou-Quisquater та Schnorr [16]. Розглянемо детальніше метод FESIS.

Його головну ідею можна сформулювати наступним чином. Спочатку користувач генерує випадковим чином два простих числа, що позначаються через p та q, після чого підраховується їх добуток $m=p\cdot q$. Наступним кроком є створення відкритого та закритого ключів. Для цього вибирається число v, що відповідає таким правилам. Для цього обраного числа v має існувати таке x, що $x^2 \bmod m = v$ і існує v^{-1} таке, що $v\cdot v^{-1} \bmod m = 1$. Далі знаходиться найменше число s таке, що рівняння $s^2 \bmod m = v^{-1}$ є правильним. Саме значення v у парі з m використовується для відкритого ключа, у той час як s потрібно для закритого ключа.

На етапі реєстрації користувач надсилає до системи пакет зі своїм відкритим ключем, тобто числа v та m.

На етапі ідентифікації користувачем генерується псевдовипадковим чином число r, а потім підраховується значення $x = r^2 \bmod m$, а отримане значення x надсилається користувачем системі. Наступним етапом є виконання t підтвердження достовірності даних користувача, кожен з яких передбачає виконання таких дій:

1. Користувачу відправляється випадково згенероване булеве значення b.

2. Якщо згенероване значення b є істинним, то користувачем відправляється до системи число r, а в іншому випадку, виконується обчислення на основі закритого ключа s значення $y = r \cdot s \bmod m$, яке після цього відправляється до системи.

3. Якщо згенероване значення b є істинним, то система обраховує $x = r^2 \bmod m$, в протилежному випадку, система обраховує $x = y^2 \cdot v \bmod m$, перевіривши при цьому, що абонент знає $s = \sqrt{v^{-1}}$.

Якщо зловмисник намагатиметься виконати спробу злому такої системи, то він не матиме доступу до секретного ключа користувача, а саме до чисел v та m. Як наслідок, якщо він зможе перехопити h циклів ідентифікації, тобто отримає інформацію з h_1 циклів при хибному значенні b та h_2 циклів при істинному значенні b, $h=h_1+h_2$. Це означає, що зловмисник матиме доступ до такої множини: $<r_i, x_i>$, $\forall i \in \{1,2,\ldots,h_1\}$: $x_i = r_i^2 \bmod m$ $<r_j, y_j>$, та $\forall j \in \{1,2,\ldots,h_2\}$: $y_j = r_j \cdot s \bmod m$. Ця множина $<r_i, x_i>$ може бути використана ним для виконання підбору значення m. З іншого боку, множину значень $<r_j, y_j>$ можна використати для підбору значення закритого ключа s. Кожна з цих двох описаних задач по своїх складності еквівалентна розкладанню числа, що має розрядність більшу за 2048, на два простих множники, що по своїх суті не є доцільним через свою складність та затрати на обчислення.

Розглянемо іншу ситуацію: нехай зловмисник отримав доступ до самої системи. Це означає, що він має доступ до відкритого ключа, тобто до значень чисел v та m. Як наслідок, він зможе згенерувати випадкове число g та підрахувати $\xi = g^2 \bmod m$ і відправити до системи ξ в якості x. Далі, у разі отримання від системи істинного значення b, зловмисник надішле у відповідь створене ним число g. Після цього система перевірить, що $\xi = g^2 \bmod m$. Але, якщо ж система відправила йому хибне значення b, то за алгоритмом зловмисник мав би надіслати до системи спеціально обчислене значення $y = g \cdot s \bmod m$, що можливо зробити тільки з використанням закритого ключа s. Оскільки він не володіє цим значенням, то і

обчислити правильне значення y йому не вдасться. Задача знаходження перебором значення закритого ключа s може бути порівняна з віднаходженням v^{-1} по відомому v. Ця задача, в свою чергу, вирішується лише якщо зловмисник має доступ до множників p і q, які разом вкладають число $m = p \cdot q$. Але, оскільки він не володіє цими множниками, то і сама задача підбору не може бути виконана у адекватний проміжок часу з адекватною кількістю обчислювальних ресурсів.

Іще одним методом, окрім методу FFSIS та його різноманітних покращень [20], що заснований на використанні концепції “нульових знань”, також широкого визнання досяг метод Guillou-Quisquater [16]. Основна ідея цього методу полягає в наступному. Спочатку користувач генерує випадковим чином пароль ϑ, що є відкритим. Іще однією складовою відкритого ключа є число m, яке формується як результат множення випадково згенерованих простих чисел p та q, які є закритими. Далі користувач знаходить такі два числа v і B, аби значення $(\vartheta \cdot B^v)$ по модулю m було рівне одиниці. На етапі реєстрації користувачу пропонується надіслати до системи два числа: його пароль ϑ та значення m.

Сам етап ідентифікації користувача можна розкласти на виконання таких дій:

1. За допомогою генератора псевдовипадкових чисел формується r.
2. Підраховується сеансовий пароль $P = r^v \bmod m$, після чого він надсилається системі.
3. Після отримання паролю P, система за допомогою генератора випадкових чисел створює число d, так щоб була правильною нерівність $0<d< m-1$; отримане число d надсилається користувачу.
4. З боку користувача обчислюється $G = r \cdot B^d \bmod m$, після чого отримане число надсилається до системи.
5. З боку системи підраховується значення $Q=G^v \cdot \vartheta^d \bmod m$. На цьому етапі приймається рішення про правильність користувача, якщо виконується рівність $Q=P$.

Можна виділити наступний ланцюжок як основу методу:

$Q = G^v \cdot \vartheta^d \bmod m = (r \cdot B^d)^v \cdot \vartheta^d \bmod m = r^v \cdot B^{d \cdot v} \cdot \vartheta^d \bmod m = r^v \cdot (\vartheta \cdot B^v)^d \bmod m = r^v \bmod m = P$.

На сьогоднішній день найбільшим недоліком цього способу є значний час його виконання, адже не тільки необхідно надіслати декілька пакетів в обидві сторони, але й виконати модульне експоненціювання на стороні користувача на кожному етапі ідентифікації.

Останнім буде розглянуто метод ідентифікації Schnorr [14], який так само як і інші базується на виборі з боку користувача такої пари простих чисел p і q, аби значення q ділило націло p-1. На наступному кроці користувач обирає таке значення для числа a, аби число a^q давало остачу 1 при діленні на p. Після цього користувач обирає згенеровано випадково число s, що має бути меншим за q: $s < q$, і буде слугувати у якості його секретного ключа. Тепер для того аби підрахувати значення відкритого ключа користувач знаходить значення $v = a^{-s}$ по модулю p, і саме він буде надісланий до системи на етапі реєстрації.

На кожному етапі проведення ідентифікації буде виконуватися такий алгоритм:

1. За допомогою генератора випадкових чисел генерується значення r таке щоб $r < q$, а далі підраховується $x = a^r \bmod p$.
2. Отримане значення x надсилається системі.
3. З боку системи створюється випадковим чином значення e, що містить h бітів, а потім надсилається до користувача.
4. Отримавши це число, з боку користувача підраховується значення $y = (r + s \cdot e)$ за модулем q, і отриманий результат надсилається системі.
5. Останнім кроком з боку системи знаходиться $\Theta = a^y \cdot v^e \bmod p$, після чого воно порівнюється з отриманим x, і у випадку їх рівності, то вважається, що користувач правильний.

Аналізуючи сильні та слабкі сторони розглянутого алгоритму, можна зробити висновок, що він заснований на використанні чисел невеликої розрядності, менших за q. Це спричиняє два важливих наслідки. З одного боку, цей факт різко зменшує обчислювальну складність алгоритму на етапі проведення ідентифікації користувача. Але з іншого боку, отриманий закритий ключ s, який за умови має бути менше q, виявляється недостатньо великим, а тому теоретично його значення не є стійким до проведення повного перебору.

Більше того, сам етап ідентифікації помітно ускладнюється та збільшується у часі через використання трьох обмінів даними між користувачем та системою, що негативно впливає на загальну картинку використання цього методу.

Підводячи підсумки по всім трьом описаним у цьому розділі алгоритмам ідентифікації користувачів, які використовують у своїх основі концепцію “нульових знань”, можна зробити висновок, що у якості бази кожен з них використовує операцію модулярного експоненціювання з використанням аргументів, бітова довжина яких набагато перевищує бітність процесора. Як наслідок, це робить такі операції достатньо складними та часо-витратними для їх виконання на процесорі, а також набагато ускладнює спроби знаходження обернених до них операцій. Проте з іншої сторони, оскільки темпи зростання бітової довжини чисел перевищують темпи зростання швидкості проведення обчислень, то відповідно збільшується і час на виконання алгоритму.

Одним із можливих удосконалень та логічним продовженням описаного механізму є використання у якості базової одиниці булевого функціонального перетворювача, попередні дослідження використання якого для виконання ідентифікації віддалених користувачів у рамках концепції “нульових знань” були опубліковані декілька років тому [17]. Можна виділити два головних недоліки їх використання. По-перше, це збільшення кількості пам’яті, яка витрачається на реалізацію методу, оскільки потрібно зберігати функціональні таблиці. По-друге,

механізм з їх використанням передбачає отримання обмеженої кількості циклів ідентифікації, яку можна буде провести перед тим як з’явиться необхідність повторно провести реєстрацію користувача в системі. І хоча кількість циклів збільшується по мірі покращення технології, вона все ж не є нескінченною. Досконалі дослідження щодо рівня захисту таких систем ще не були проведені повною мірою.

Висновки до розділу 1

Як результат проведеного аналізу існуючих досліджень у сфері способів та механізмів ідентифікації віддалених користувачів, зокрема у системах багатьох користувачів, а також ретельного огляду та порівняння існуючих протоколів ідентифікації, що використовуються на практиці, можна сформулювати такі висновки:

1. Головну слабку сторону систем, які базуються на використанні паролю користувача при виконання його ідентифікації, можна описати наступним чином. Оскільки такий механізм порушує головне правило захисту інформації, а саме принцип, що завжди має існувати лише один власник секретних даних, то системи з використанням лише паролю користувача не є надійними і є вразливими до різноманітних атак як на саму систему, так і на користувача, чи навіть канал зв’язку між користувачем і системою. Злам будь-якої з цих ланок призводить до отримання легкого доступу до всіх даних користувача.

2. Беручи до уваги висновок першого пункту, найбільш перспективним у теоретичному плані можна назвати механізми ідентифікації, які беруть за свою основу принципи концепції “нульових знань”.

3. Методи виконання криптографічно-строгої ідентифікації на основі концепції “нульових знань”, що використовуються на сьогоднішній день, базуються на математичній операції модулярного експоненціювання, яка є теоретично незворотньою. Через значну кількість обчислювальних ресурсів, які необхідно витрачати на проведення таких операції, популярні алгоритми зазвичай зменшують розрядність чисел, або використовують малі значення степеню, до якого підносяться числа. Аби відіграти втрати у захищеності при використання таких поступок, зазвичай використовують декілька циклів обміну інформацією між системою та користувачем. Як наслідок, збільшується час, необхідний на

виконання ідентифікації, що погано впливає на функціонування усієї системи загалом. Це є основним недоліком існуючих алгоритмів.

4. Базуючись на висновку, сформульованому у попередньому пункті, можна сказати, що головними напрямками покращення швидкодії та ефективності ідентифікації користувачів загалом є, по-перше, зменшення кількості циклів обміну інформацією між користувачем та системою, і по-друге, перехід до альтернативного алгебраїчного базису для алгоритму ідентифікації, який дозволить прискорити ідентифікацію за рахунок можливості її реалізації спеціальними апаратними засобами, а також знизить кількість обчислювальних ресурсів, які необхідні на виконання однієї ітерації механізму ідентифікації.

РОЗДІЛ 2

РОЗРОБКА МЕТОДУ ІДЕНТИФІКАЦІЇ НА ОСНОВІ НЕЗВОРОТНИХ БУЛЕВИХ ПЕРЕТВОРЕНЬ

2.1. Теоретичне обґрунтування використання незворотних перетворень для криптографічно-строгої ідентифікації

Вирішення сформульованої у першому розділі задачі, а саме значне збільшення швидкості виконання ідентифікації у системі, може бути досягнуто завдяки переходу до іншого алгебраїчного базису в механізмі проведення ідентифікації. У якості такого базису були обрані булеві функціональні перетворювачі з нелінійною залежністю.

Основною якістю, якою володіють булеві нелінійні функціональні перетворювачі у контексті криптографічно-строгої ідентифікації, є незворотність перетворення, що вони виконують. Під незворотністю перетворення мається на увазі, що застосування перетворення в одну сторону — у прямому ході — є досить легким та очевидним, у той час як знаходження оберненого перетворення, тобто пошук такого значення вхідного вектору, яке дало б заданий вихідний вектор після застосування до нього перетворення, є надто складною і близькою до неможливою задачею.

Другим важливим фактором таких булевих функціональних перетворювачів є їх відносно легка апаратна реалізація через те, що вони зазвичай використовують базові булеві операції для роботи. Це дає їм значну перевагу, і вважається, вони здатні забезпечити прискорення на декілька порядків при принаймні тому самому рівні захисту відносно інших алгебраїчних базисів.

Через це сучасні методи криптографічного захисту даних, як-от різноманітні хеш-перетворення, потокові шифри та алгоритми симетричного шифрування, в основному використовують саме булеві функціональні перетворювачі.

Згідно проведеного аналізу, булевий функціональний перетворювач, що розробляється, має задовольняти таким вимогам:

1. перетворювач має мати достатньо просту структуру для ефективної апаратної та програмної реалізації;

2. при цьому він має мати можливість забезпечувати формування у рамках одного дизайну безлічі відмінних між собою функціональних перетворювачів;

3. зважаючи на бітову довжину чисел, які будуть використовуватися, дизайн функціонального перетворювача повинен бути представлений у процедурній формі;

4. перетворення, що будуть використовуватися, мають бути нелінійними для забезпечення стійкості до лінійного криптоаналізу;

5. для підвищення криптографічної надійності кожен з бітів вихідного числа має мати функціональну залежність від усіх бітів вхідного числа.

Аналіз показав, що поставлена задача проведення швидкої ідентифікації з належним рівнем криптографічного захисту може бути вирішена з використанням хеш-перетворень, для яких колізії задаються та програмуються на етапі побудови самого перетворювача.

Поставлена задача побудови такого перетворювача еквівалентна створенню нелінійної функції з властивістю незворотності, яка буде мати значень вхідного вектору, кожне з яких при застосуванні до нього функціонального перетворення буде давати одне й те саме значення вихідного вектору.

Сформульованим вище вимогам відповідає зображена на рис. 2.1 архітектура булевого функціонального перетворювача, що для наочності показана для конкретного випадку розбиття вихідного вектору на три фрагменти.

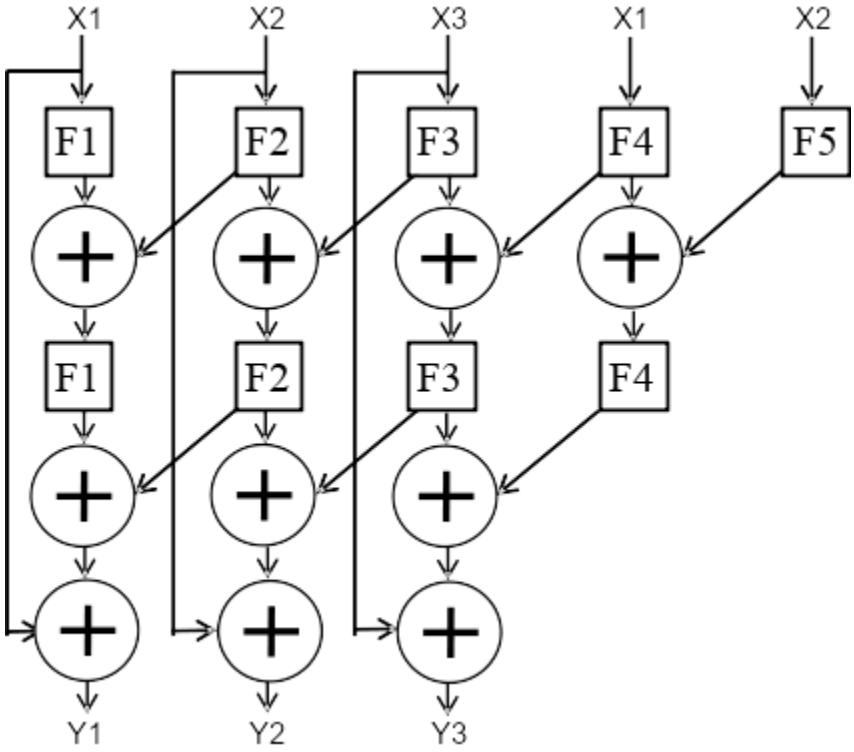

Рис. 2.1. Архітектура булевого функціонального перетворювача при розбитті вихідного вектору на три фрагменти

При описі архітектури функціонального перетворювача використовуються такі умовні позначення:

1. k – бітова довжина одного фрагмента вхідного та вихідного вектору;
2. n – загальна бітова довжина вихідного вектору;
3. $m = \frac{n}{k}$ – кількість фрагментів, на які розбивається вихідний вектор;
4. $p=2\,m - 1$ – кількість фрагментів, з яких складається вхідний вектор;
5. $s=k\,p$ – загальна бітова довжина вхідного вектору;
6. $X = \{x_1, x_2 \ldots x_p\}$ – множина фрагментів, що утворює вхідний вектор, $\forall\, i = 1,2 \ldots p : x_i \in \{0,1,\ldots,2^k - 1\}$;
7. $Y = \{y_1, y_2 \ldots y_m\}$ – множина фрагментів, що утворює вихідний вектор, $\forall\, i = 1,2 \ldots m : y_i \in \{0,1,\ldots,2^k - 1\}$.

Архітектура містить *p* різних функціональних перетворювачів (позначених F_i), кількість яких рівна кількості фрагментів вхідного вектору. Кожен з цих перетворювачів представляє собою так звану "*look-up table*", де для кожного вхідного значення вказується значення, яке подається на вихід перетворювача. Таким чином, функціональний перетворювач можна подати за допомогою таблиці, яка складається з 2^k рядків, де кожен рядок містить комірку з певним значенням, і кожній комірці однозначно відповідає її адреса — порядковий номер рядка у таблиці. Тут і далі під вхідним значенням функціонального перетворювача буде вважатися адреса комірки, яку потрібно взяти, а числове значення у комірці за цією адресою буде вихідним значенням з функціонального перетворювача.

Оскільки архітектура передбачає використання декількох шарів функціональних перетворювачів, то для значення на вхід функціонального перетворювача під номером j на і-му шарі вводиться позначення $v_{i,j}$, а для його значення на виході — $w_{i,j}$, отже загалом має місце рівність $f_j(v_{i,j}) = w_{i,j}$.

Загальна робота булевого функціонального перетворювача може бути як формування з вхідного вектору *X*, що має бітову довжину s, вихідного вектору *Y* довжини *n*. При цьому за смисловим значенням вхідні вектори *X* є сеансовими паролями для ідентифікації користувача, а вектори *Y* — ідентифікаторами користувача.

Загальний процес проходження ідентифікації у системі за допомогою запропонованого булевого функціонального перетворювача можна поділити на два ключових етапи:

1. користувач реєструється у системі;

2. користувач ідентифікується у системі.

У свою чергу, етап реєстрації користувача у системі передбачає виконання таких кроків:

1. користувач формує запит на реєстрацію на надсилає його до системи;

2. отримавши запит на реєстрацію, система генерує новий унікальний ідентифікатор користувача Y. Цей згенерований ідентифікатор відправляється у відповідь користувачу;

3. отримавши від системи свій новий персональний ідентифікатор Y, користувач за допомогою запропонованого далі методу генерує булеве функціональне перетворення з програмованими колізіями, та отримує на вихід власну множину згенерованих паролів X;

4. згенерована множина паролів X зберігається користувачем у власному захищеному розділі пам’яті;

5. отримане у процесі генерації булеве функціональне перетворення, а саме значення таблиць функціональних перетворювачів, відправляються до системи;

6. отримавши булеве функціональне перетворення від користувача, система зберігає його для цього користувача разом із його ідентифікатором Y.

Етап ідентифікації користувача q-тий раз у системі передбачає виконання таких кроків:

1. користувач дістає зі своєї збереженої на етапі реєстрації множини паролів черговий невикористаний пароль X_q;

2. цей пароль X_q надсилається користувачем до системи для проходження ідентифікації;

3. отримавши запит на ідентифікацію від користувача з паролем X_q, система застосовує збережене для цього користувача булеве функціональне перетворення до отриманого пароля X_q та отримує значення Y_q;

4. отримавши значення Y_q, система порівнює його зі збереженим ідентифікатором цього користувача Y, і якщо це значення співпадає з отриманим Y_q, то вважається, що користувач успішно пройшов ідентифікацію, і пароль X_q додається до списку використаних паролів цього користувача. Інакше, якщо

значення Y не співпадає з отриманим Y_q, то вважається, що користувач ідентифікацію в системі не пройшов.

2.2. Розробка процедури побудови перетворення

Далі буде детально описано алгоритм, який може бути застосований до запропонованої архітектури булевого функціонального перетворювача для заповнення таблиць його функціональних перетворювачів і одночасного формування множини вхідних векторів X таких, що якщо застосувати отримане перетворення до будь-якого з них, на виході перетворювача буде отримано одне і те саме значення вихідного вектору Y.

Для роботи алгоритм отримую на вхід значення вихідного вектору Y, для якого потрібно згенерувати перетворення та множину вхідних векторів X.

Запропонований алгоритм складається з таких кроків:

1. Для кожного функціонального перетворювача F_j ініціалізувати усі його комірки невизначеним станом, який буде позначатися через (*-1*): $\forall j = 1,2..2m-1, l = 0,1..2^k - 1: f_j(l) = -1$.

2. Встановити номер поточного шару у значення *m-1*, що відповідає останньому шару: *i=m-1*; ініціалізувати номер q поточного вектору X, що буде згенеровано: *q=1*.

3. У кожному функціональному перетворювачі F_j поточного шару за допомогою генератора псевдовипадкових чисел знайти адресу $v_{i,j}$ комірки, яка містить невизначений стан: $\forall j = 1..m+1 : f_j(v_{i,j}) = -1$.

4. У якості поточного шару вибрати попередній шар: $i = i - 1$.

5. За допомогою генератора псевдовипадкових чисел вибрати таку $v_{i,1}$ адресу комірки поточного шару, значення якої у першому функціональному перетворювачі є визначеним: $f_1(v_{i,1}) \neq -1$; якщо не існує такої комірки у першому функціональному перетворювачі з визначеним станом, то згенерувати її адресу $v_{i,1}$ та значення $w_{i,1}$ за допомогою генератора випадкових чисел так, аби її адреса $v_{i,1}$

не була рівною вибраній в пункті 3 адресі $v_{m-1,1}$: $v_{i,1} \neq v_{m-1,1}$, і записати до F_1 за адресою $v_{i,1}$ значення $w_{i,1}$: $f_1(v_{i,1}) = w_{i,1}$.

6. Значення на виході $w_{i,j}$ для решти функціональних перетворювачів поточного шару $(j = 1,2..2m - i)$ визначається послідовно за наступною формулою: $w_{i,j} = w_{i,j-1} \oplus v_{i+1,j-1}$; відповідні адреси $v_{i,j}$ комірок для кожного з F_j обираються за допомогою генератора випадкових чисел так, аби вміст цієї комірки був рівний знайденому $w_{i,j}$: $f_j(v_{i,j}) = w_{i,j}$; якщо не існує комірок з таким значенням $w_{i,j}$, то за допомогою генератора випадкових чисел обирається будь-яка комірка поточного функціонального перетворювача F_j, стан якої невизначений, а адреса $v_{i,j}$ якої не рівна вибраній в пункті 3 $v_{m-1,j}$: $v_{i,j} \neq v_{m-1,j}$, після чого до цієї комірки записується підраховане значення $w_{i,j}$: $f_j(v_{i,j}) = w_{i,j}$.

7. Якщо поточний шар не є першим, тобто якщо його номер $i \geq 2$, то перейти до виконання пункту 4, інакше перейти до пункту 8.

8. Оскільки поточний шар є першим шаром, то відповідні фрагменти поточного вхідного вектору X_q мають бути рівними відповідним значенням на вході до функціональних перетворювачів поточного шару, і тому їх значення визначаються як: $x_{q,j} = v_{i,1}$ $(j = 1,2..2m - 1)$.

9. За допомогою генератора випадкових чисел згенерувати вміст $w_{m-1,1}$ комірки на останньому шарі першого функціонального перетворювача F_1, адреса $v_{m-1,1}$ якої була визначена у пункті 3, і записати значення $w_{m-1,1}$ за адресою $v_{m-1,1}$ до першого функціонального перетворювача: $f_1(v_{m-1,1}) = w_{m-1,1}$.

10. Значення на виході $w_{m-1,j}$ решти функціональних перетворювачів F_j в останньому шарі $(j = 2,3..m + 1)$.обчислити послідовно за наступною формулою: $w_{m-1,j}=(x_{q,j-1} \oplus y_{j-1}) \oplus w_{m-1,j-1}$, а відповідні комірки функціональних перетворювачів заповнити за адресою $v_{m-1,j}$, визначеною у пункті 3: $f_j(v_{m-1,j}) = w_{m-1,j}$.

11. Знайти кількість u_j комірок з невизначеним значенням у кожному функціональному перетворювачі F_j $(j = 1,2..2m - 1)$.

12. Якщо кількість u_j комірок з невизначеним значенням у всіх функціональних перетворювачах F_j більша або рівна *m-1*: $\forall j \in \{1,2,\dots,2m-1\}$: $u_j \geq m-1$, то збільшити на 1 номер поточного вхідного вектора: *q=q+1*, і перейти до пункту 2; інакше перейти до пункту 13.

13. За допомогою генератора випадкових чисел для кожного функціонального перетворювача визначити значення усіх комірок з невизначеним значенням числами у діапазоні $0..2^k-1$.

2.3. Приклад виконання запропонованої процедури побудови перетворення

Для ілюстрації роботи запропонованого алгоритму наводиться приклад реєстрації користувача з ідентифікатором (значенням вихідного вектору) $Y=\{4,1,7\}$. Буде наведено покрокове виконання алгоритму з генерацією паролів користувача (вхідних векторів *X*) та заповненню функціональних перетворювачів.

Буде використано булевий функціональний перетворювач з параметрами $k=3$ та $m=3$, зображений на рис. 2.2. На рисунку та в таблицях порожніми комірками позначено комірки, що мають невизначений стан (*-1*).

Генерація першого паролю користувача X_1.

Усі комірки усіх функціональних перетворювачів ініціалізуються порожнім значенням, що відповідає невизначеному стану (в алгоритмі такому стану відповідає значення комірки *-1*).

У відповідності до другого та третього пункту алгоритму, обирається останній шар ($i=m-1=2$), і для кожного функціонального перетворювача в цьому шарі шукається будь-яка порожня комірка. Для прикладу візьмемо комірки з адресами $v_{2,1}=5, v_{2,2}=2, v_{2,3}=6, v_{2,4}=3$.

Згідно четвертого пункту, переходимо до попереднього шару: $i=i-1=1$.

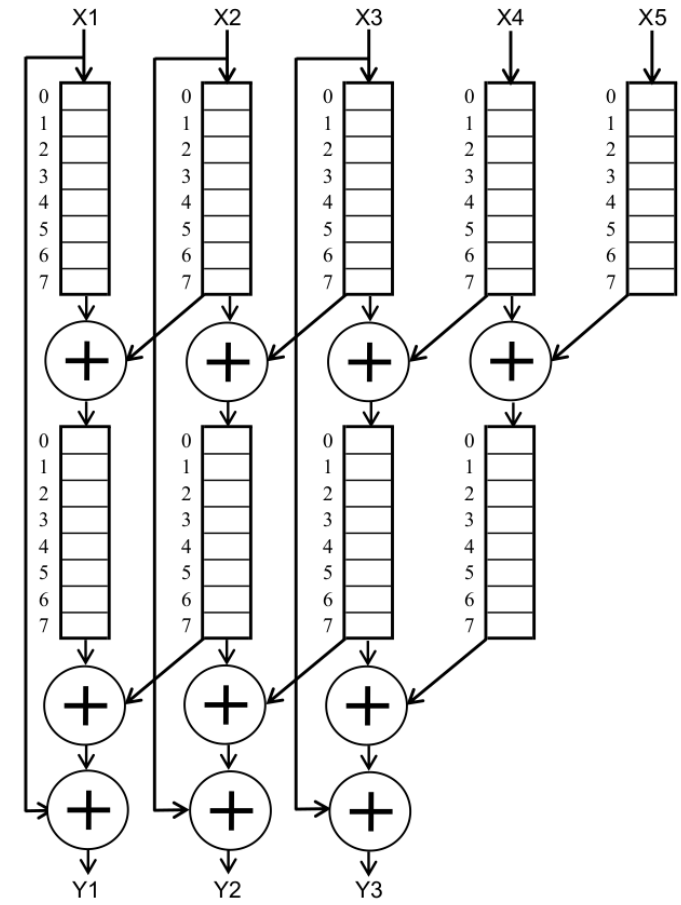

Рис. 2.2 Структура з параметрами $k = 3$ та $m = 3$

Згідно п’ятого пункту, оскільки перший функціональний перетворювач містить лише порожні комірки, то обираємо будь-яку з них. Для прикладу, візьмемо комірку з адресою $v_{1,1} = 6$ і згенеруємо для неї значення $w_{1,1} = 2$, а до таблиці F_1 занесемо $f_1(6) = 2$.

Згідно шостого пункту, знаходимо значення на виході $w_{1,j}$ решти функціональних перетворювачів у поточному шарі. Відповідні значення обчислюються як: $w_{1,2} = 5 \oplus 2 = 7, w_{1,3} = 2 \oplus 7 = 5, w_{1,4} = 6 \oplus 5 = 3, w_{1,5} = 3 \oplus 3 = 0$. Оскільки на даний момент ці функціональні перетворювачі зберігають лише порожні комірки, то відповідні адреси для цих комірок обираються випадковим чином. Для прикладу,

нехай було обрано адреси $v_{1,2}=3, v_{1,3}=4, v_{1,4}=2, v_{1,5}=7$. Обрані значення записуються до функціональних перетворювачів: $f_2(3)=7, f_3(4)=5, f_4(2)=3, f_5(7)=0$.

Оскільки поточним шаром є шар з номером один, то переходимо до пункту 8 алгоритму.

Значення фрагментів першого вхідного вектору X_1 рівні відповідним значенням на вході до функціональних перетворювачів, тобто $x_{1,1}=v_{1,1}=6, x_{1,2}=v_{1,2}=3, x_{1,3}=v_{1,3}=4, x_{1,4}=v_{1,4}=2, x_{1,5}=v_{1,5}=7$.

Відповідно до дев'ятого пункту, для першого функціонального перетворювача в останньому шарі обирається випадкове значення комірки $w_{2,1}=4$ з адресою комірки $v_{2,1}=5$, що була знайдена у третьому пункті. Це значення заноситься до таблиці першого функціонального перетворювача: $f_1(5)=4$.

Згідно десятого пункту алгоритму, знаходимо решту значень на виході з функціональних перетворювачів для останнього шару за допомогою формули $w_{m-1,j}=(x_{q,j-1}\oplus y_{j-1})\oplus w_{m-1,j-1}$ (значення на вході до них були обрані у третьому пункті): $w_{2,2}=(6\oplus 4)\oplus 4=6, w_{2,3}=(3\oplus 1)\oplus 6=4, w_{2,4}=(4\oplus 7)\oplus 4=7$. Ці значення заносяться до таблиць відповідних функціональних перетворювачів: $f_2(2)=6, f_3(6)=4, f_4(3)=7$.

Завершено генерування першого паролю користувача, він має значення: $X_1=\{6,3,4,2,7\}$. Значення таблиць функціональних перетворювачів після виконання першої ітерації алгоритму наведено у таблиці 2.1 зліва (порожні комірки відповідають невизначеному стану).

Згідно одинадцятого та дванадцятого пункту алгоритму, оскільки у кожному функціональному перетворювачі залишилося принаймні 2 порожні комірки, то можна продовжити генерацію паролів користувача.

Генерація другого паролю користувача X_2.

Відповідно другого та третього пункту алгоритму, потомним шаром обираємо останній $i=m-1=2$, і для усіх функціональних перетворювачів

знаходимо порожню комірку у кожному з них. Для прикладу візьмемо комірки з адресами $v_{2,1} = 0, v_{2,2} = 5, v_{2,3} = 1, v_{2,4} = 6$.

Далі переходимо до попереднього шару: $i = i - 1 = 1$.

Таблиця 2.1

Значення таблиць функціональних перетворювачів
після 1-ї (зліва) та 2-ї (справа) ітерації алгоритму

Значення за адресою	Функціональний перетворювач				
	F_1	F_2	F_3	F_4	F_5
v_1					
v_2					
v_3		6		3	
v_4		7		7	
v_5			5		
v_6	4				
v_7	2		4		
v_8					0

Значення за адресою	Функціональний перетворювач				
	F_1	F_2	F_3	F_4	F_5
v_1	5				
v_2			1		
v_3		6		3	
v_4		7	7	7	
v_5			5	6	
v_6	4	7			
v_7	2		4	5	
v_8		2			0

Згідно п’ятого пункту, у першому функціональному перетворювачів обирається комірка, що не є порожньою. Для прикладу, візьмемо комірку зі значенням $w_{1,1} = 2$ за адресою $v_{1,1} = 6$.

Послідовно для всіх інших функціональних перетворювачів в поточному шарі обчислюємо вихідні значення згідно формули $w_{i,j} = (w_{i,j-1} \oplus v_{i+1,j-1})$. Отримуємо такі значення: $w_{1,2} = 2 \oplus 0 = 2, w_{1,3} = 2 \oplus 5 = 7, w_{1,4} = 7 \oplus 1 = 6, w_{1,5} = 6 \oplus 6 = 0$. Оскільки тільки один з функціональних перетворювачів містить комірку з відповідними значенням (F_5), то адреси для інших комірок $v_{1,j}$ будуть обрані за допомогою генератора

випадкових чисел. Для прикладу, візьмемо значення адрес $v_{1,2} = 7, v_{1,3} = 3, v_{1,4} = 4$ і занесемо остаточні значення до таблиць функціональних перетворювачів: $f_2(7) = 2, f_3(3) = 7, f_4(4) = 6$.

Оскільки поточний шар є першим шаром (з номером 1), то переходимо до виконання восьмого пункту і прирівнюємо значення другого вхідного вектору до відповідних значень на вхід до функціональних перетворювачів на першому шарі: $x_{2,1} = v_{1,1} = 6, x_{2,2} = v_{1,2} = 7, x_{2,3} = v_{1,3} = 3, x_{2,4} = v_{1,4} = 4, x_{2,5} = v_{1,5} = 7$.

Переходимо до дев'ятого пункту і генеруємо випадкове значення для комірки першого функціонального перетворювача за адресою $v_{2,1} = 0$, яка була визначена в третьому пункті. Для прикладу візьмемо $w_{2,1} = 5$ і занесемо результат до першого функціонального перетворювача: $f_1(0) = 5$.

Далі послідовно обраховуємо решту значень на виході функціональних перетворювачів на останньому шарі згідно формули $w_{\text{m-1,j}} = (\text{x}_{\text{q,j-1}} \oplus \text{y}_{\text{j-1}}) \oplus \text{w}_{\text{m-1,j-1}}$, а саме: $w_{2,2} = (6 \oplus 4) \oplus 5 = 7, w_{2,3} = (7 \oplus 1) \oplus 7 = 1, w_{2,4} = (3 \oplus 7) \oplus 1 = 5$. Заносимо отримані значення до комірок за адресами з третього пункту: $f_2(5) = 7, f_3(1) = 1, f_4(6) = 5$.

На цій ітерації було згенеровано другий пароль користувача зі значенням $X_2 = \{6,7,3,4,7\}$. Значення таблиць функціональних перетворювачів після виконання другої ітерації алгоритму наведено у таблиці 2.1 справа (порожні комірки відповідають невизначеному стану).

Згідно одинадцятого та дванадцятого пункту алгоритму, оскільки у кожному функціональному перетворювачі залишилося принаймні 2 порожні комірки, то можна продовжити генерацію паролів користувача.

Генерація третього паролю користувача X_3.

Відповідно до другого та третього пунктів алгоритму, поточним шаром є останній $(i = m - 1 = 2)$. Знаходимо в цьому шарі адреси пустих комірок. Для прикладу візьмемо адреси $v_{2,1} = 2, v_{2,2} = 1, v_{2,3} = 7, v_{2,4} = 0$.

Переходимо до попереднього шару: $i = i - 1 = 1$.

Згідно п’ятого пункту, знаходимо будь-яку комірку першого функціонального перетворювача, що не є порожньою. Для прикладу візьмемо комірку, що має адресу $v_{1,1} = 0$ і містить число $w_{1,1} = 5$.

Відповідно до шостого пункту алгоритму, за допомогою формули $w_{i,j} = (w_{i,j-1} \oplus v_{i+1,j-1})$ знаходимо значення на виході з решти функціональних перетворювачів поточного шару: $w_{1,2} = 5 \oplus 2 = 4, w_{1,3} = 4 \oplus 1 = 5, w_{1,4} = 5 \oplus 7 = 2, w_{1,5} = 2 \oplus 0 = 2$. Оскільки третій функціональний перетворювач містить комірку з таким значеннями, то ми обираємо саме її, тому адреса буде $v_{1,3} = 4$. Так як у другого, четвертого та п’ятого функціонального перетворювача комірок із такими значеннями немає, то адресу для них обираємо будь-яку серед порожніх комірок. Для прикладу візьмемо $v_{1,2} = 4, v_{1,4} = 1, v_{1,5} = 3$ і нанесемо ці значення до таблиць функціональних перетворювачів: $f_2(4) = 4, f_4(1) = 2, f_5(3) = 2$.

Ми знаходимось на першому шарі, тому переходимо до восьмого пункту алгоритму і прирівнюємо значення фрагментів вхідного вектору (третього паролю користувача) до значень на вході до відповідних функціональних перетворювачів першого шару: $x_{3,1} = v_{1,1} = 0, x_{3,2} = v_{1,2} = 4, x_{3,3} = v_{1,3} = 4, x_{3,4} = v_{1,4} = 1, x_{3,5} = v_{1,5} = 3$.

На дев’ятому пункті алгоритму генеруємо випадкове значення для комірки першого функціонального перетворювача з адресою $v_{2,1} = 2$. Для прикладу візьмемо значення $w_{2,1} = 3$ та занесемо його до таблиці функціонального перетворювача: $f_1(2) = 3$.

Згідно десятого пункту, по формулі $w_{m-1,j} = (x_{q,j-1} \oplus y_{j-1}) \oplus w_{m-1,j-1}$ обчислюємо решту значень на виході з функціональних перетворювачів останнього шару по адресам, обраним у третьому пункті:

$w_{2,2} = (0 \oplus 4) \oplus 3 = 3, w_{2,3} = (4 \oplus 1) \oplus 3 = 6, w_{2,4} = (4 \oplus 7) \oplus 6 = 5$, після чого заносимо їх до таблиць функціональних перетворювачів: $f_2(1) = 3, f_3(7) = 6, f_4(0) = 5$.

На цій ітерації було згенеровано третій пароль користувача зі значенням $X_3 = \{0,4,4,1,3\}$. Значення таблиць функціональних перетворювачів після виконання третьої ітерації алгоритму наведено у таблиці 2.2 зліва (порожні комірки відповідають невизначеному стану).

Згідно одинадцятого та дванадцятого пункту алгоритму, оскільки у кожному функціональному перетворювачі залишилося принаймні 2 порожні комірки, то можна продовжити генерацію паролів користувача.

Таблиця 2.2

Значення таблиць функціональних перетворювачів
після 3-ї (зліва) та 4-ї (справа) ітерації алгоритму

Значення за адресою	Функціональний перетворювач				
	F_1	F_2	F_3	F_4	F_5
v_1	5			5	
v_2		3	1	2	
v_3	3	6		3	
v_4		7	7	7	2
v_5		4	5	6	
v_6	4	7			
v_7	2		4	5	
v_8		2	6		0

Значення за адресою	Функціональний перетворювач				
	F_1	F_2	F_3	F_4	F_5
v_1	5	5		5	
v_2		3	1	2	
v_3	3	6		3	
v_4		7	7	7	2
v_5	7	4	5	6	
v_6	4	7	6		4
v_7	2		4	5	
v_8		2	6	6	0

Генерація четвертого паролю користувача X_4.

Відповідно другого та третього пунктів алгоритму, шар під номером два обирається поточним: $i = m - 1 = 2$. У кожному функціональному перетворювачі

останнього (другого) шару шукаємо комірку з порожнім значенням. Для прикладу візьмемо такі адреси комірок: $v_{2,1} = 4, v_{2,2} = 0, v_{2,3} = 5, v_{2,4} = 7$.

Переходимо до попереднього шару під номером один і вибираємо у першому функціональному перетворювачі будь-яку комірку, що не є порожньою. Для прикладу візьмемо комірку зі значенням $w_{1,1} = 2$ за адресою $v_{1,1} = 6$.

Відповідно до шостого пункту алгоритму, з використанням формули $w_{i,j} = (w_{i,j-1} \oplus v_{i+1,j-1})$ знаходяться решта значень на виході функціональних перетворювачів першого шару: $w_{1,2} = 2 \oplus 4 = 6, w_{1,3} = 6 \oplus 0 = 6, w_{1,4} = 6 \oplus 5 = 3, w_{1,5} = 3 \oplus 7 = 4$. Оскільки функціональні перетворювачі 2, 3 та 4 містять комірки зі значеннями $w_{1,2} = 6$, $w_{1,3} = 6$ та $w_{1,4} = 3$, відповідно, то обираються саме вони, а отже адреси матимуть значення $v_{1,2} = 2, v_{1,3} = 7, v_{1,4} = 2$. У п'ятого функціонального перетворювача немає комірки, яка б містила число $w_{1,5} = 4$, тому візьмемо випадкову порожню комірку, нехай це буде комірка за адресою $v_{1,5} = 5$, і занесемо до неї це значення: $f_5(5) = 4$.

Відповідно до сьомого пункту алгоритму, переходимо до восьмого пункту і прирівнюємо фрагменти вхідного вектору (четвертого паролю користувача) до відповідних значень на вході функціональних перетворювачів у першому шарі: $x_{4,1} = v_{1,1} = 6, x_{4,2} = v_{1,2} = 2, x_{4,3} = v_{1,3} = 7, x_{4,4} = v_{1,4} = 2, x_{4,5} = v_{1,5} = 5$.

Відповідно до дев'ятого пункту алгоритму, для першого функціонального перетворювача обираємо випадкове значення комірки за адресою $v_{2,1} = 4$. Для прикладу візьмемо значення $w_{2,1} = 7$ і занесемо його до таблиці першого функціонального перетворювача: $f_1(4) = 7$.

Згідно десятого пункту алгоритму, використовуючи формулу $w_{m-1,j} = (x_{q,j-1} \oplus y_{j-1}) \oplus w_{m-1,j-1}$ підраховуємо решту значень на виході функціональних перетворювачів на останньому (другому) шарі, а саме:

$w_{2,2} = (6 \oplus 4) \oplus 7 = 5, w_{2,3} = (2 \oplus 1) \oplus 5 = 6, w_{2,4} = (7 \oplus 7) \oplus 6 = 6$, після чого заносимо ці значення до таблиць функціональних перетворювачів: $f_2(0) = 5, f_3(5) = 6, f_4(7) = 6$.

На цій ітерації алгоритму було згенеровано значення четвертого паролю користувача, а саме $X_4 = \{6,2,7,2,5\}$. Значення таблиць функціональних перетворювачів після виконання третьої ітерації алгоритму наведено у таблиці 2.2 справа (порожні комірки відповідають невизначеному стану).

Оскільки, як видно з таблиці, у третьому функціональному перетворювачі залишилася лише одна комірка (дванадцятий пункт алгоритму), що містить порожнє значення, то на цьому генерація паролів користувача завершується і алгоритм переходить до тринадцятого пункту.

Згідно тринадцятого пункту алгоритму, усі комірки усіх таблиць функціональних перетворювачів, які містять порожнє значення, заповнюються випадковими значеннями, згенерованими з діапазону $0 \ldots 2^k - 1$.

Отже, завдяки роботі алгоритму було створено булеве функціональне перетворення для користувача з ідентифікатором $Y = \{4,1,7\}$ та згенеровано чотири паролі для нього: $X_1 = \{6,3,4,2,7\}$, $X_2 = \{6,7,3,4,7\}$, $X_3 = \{0,4,4,1,3\}$ та $X_4 = \{6,2,7,2,5\}$ таких, що якщо застосувати до них згенероване булеве функціональне перетворення, то на виході буде отримано ідентифікатор $Y = \{4,1,7\}$.

2.4. Теоретична та експериментальна оцінка характеристик та ефективності розробленого методу

Для перевірки ефективності роботи запропонованого алгоритму було проведено його ретельне тестування для різних вхідних параметрів за допомогою спеціально розробленого програмного застосунку.

Проведене тестування дає змогу оцінити приблизну кількість паролів користувача X, які можуть бути згенеровані за допомогою запропонованого

алгоритму для заданого ідентифікатора користувача Y розрядності n при різній розрядності k фрагментів вхідного та вихідного векторів.

Опис розробленого програмного застосунку, а також вказівки до його використання, наведені у третьому розділі.

Кількість згенерованих паролів користувача, знайдена за допомогою створеного програмного застосунку, наведена у таблиці 2.3, і знаходиться на перетині відповідних значень кількості фрагментів *m*, на які розбивається вихідний вектор (ідентифікатор користувача) та бітової довжини *k* кожного фрагменту вхідного та вихідного векторів.

Таблиця 2.3

Кількість згенерованих паролів користувача X

m кількість фрагментів векторів	Бітова довжина одного фрагменту *k*							
	11	12	13	14	15	16	17	18
5	821	1619	3355	6592	13202	26317	52484	104879
6	689	1389	2801	5587	11318	22425	44954	90050
7	615	1242	2464	4941	9913	19812	39532	79204
8	528	1105	2214	4423	8861	17739	35534	71068
9	497	996	2011	3994	8019	16155	32259	64598
10	452	908	1839	3692	7343	14813	29560	59364
11	420	842	1681	3411	6847	13649	27442	54995

Висновки до розділу 2

У другому розділі було описано новий метод виконання криптографічно-строгої ідентифікації користувачів, а також проведено тестування його ефективності. Можна виділити такі ключові висновки:

1. Було сформульовано вимоги до булевого функціонального перетворювача, який має задовольняти вимогам криптографічної строгості.
2. Була запропонована архітектура булевого функціонального перетворювача, що відповідає поставленим вимогам, та використовує хеш-перетворення з можливістю програмування колізій на етапі генерації перетворення.
3. Був запропонований алгоритм конструювання такого перетворення, що програмує в ньому колізії та генерує паролі для користувачів.
4. Робота алгоритму ретельно описана за допомогою покрокового прикладу його виконання.
5. Проведене за допомогою створеного програмного застосунку для різних розмірностей вхідних даних тестування описаного алгоритму продемонструвало, що описаний алгоритм здатен забезпечити генерацію значної кількості паролів за невеликий проміжок часу, що свідчить про його ефективність та можливість застосування для реальних потреб.

РОЗДІЛ 3

РОЗРОБКА ЗАСОБІВ ПІДВИЩЕННЯ КРИПТОСТІЙКОСТІ БУЛЕВИХ ПЕРЕТВОРЕНЬ ДЛЯ ІДЕНТИФІКАЦІЇ

3.1. Лавинний ефект та стійкість до диференційного криптоаналізу

Запропонований у другому розділі метод побудови хеш-перетворення передбачає необхідність довизначення згенерованих функціональних перетворень для наборів, значення комірок яких залишилися порожніми після виконання алгоритму. Таким чином, постає задача довизначити отриману функцію так, аби максимізувати її стійкість до криптоаналізу.

При оцінці криптостійкості функцій найбільш важливим з точки зору практичного застосування є властивість так званого лавинного ефекту (Strict Avalanche Criterion), яке характеризується максимальним значенням диференціальної ентропії зміни значення функції при інвертуванні значення будь-якої вхідної змінної, для яких функція є визначеною. Булеві функції, які володіють властивістю строгого лавинного ефекту, грають ключову роль при створенні широкого класу алгоритмів захисту інформації, оскільки ця властивість забезпечує стійкість до порушення захисту диференціальним або лінійним криптоаналізом. На практиці отримання функцій, що володіють строгим лавинним ефектом, є важливою та складною задачею.

Булева функція $f(x_1, \dots, x_n)$ від n змінних є визначеною на множині 2^n наборів і приймає значення з множини $\{0,1\}$ на кожному з них.

Булева функція вважається балансною, якщо вона приймає значення нуля та одиниці з однаковою ймовірністю. Також кажуть, що балансна функція відповідає критерію максимуму повної ентропії. Для балансної булевої функції виконується така рівність: $\sum_{x_1,\dots,x_n \in Z} f(x_1, \dots, x_n) = 2^{n-1}$.

Булева функція $f(x_1, \dots, x_n)$ відповідає строгому лавинному критерію (SAC), якщо зміна значення будь-якої з її вхідних змінних призводить до зміни вихідного значення самої функції з ймовірністю 0.5. Для SAC функції виконується така рівність: $\sum_{x_1,\dots,x_n \in Z} f(x_1, \dots x_j \dots, x_n) \oplus f(x_1, \dots \overline{x_j} \dots, x_n) = 2^{n-1}$.

При створенні балансних SAC-функцій з практичної точки зору потрібно враховувати питання обчислювальних ресурсів, що витрачаються в процесі генерації таких функцій. Загалом, питання кількості балансних SAC-функцій, що можуть бути згенеровані від n змінних залишається відкритим на сьогоднішній день [15]. Зважаючи на важливість вирішення цієї задачі, за останні два десятки років були запропоновані різні методики підвищення ефективності автоматизації генерації таких функцій. Усі відомі на сьогоднішній день методи їх генерації зводяться до створення відповідних функцій «з нуля», тобто виходять із того, що не існує додаткових обмежень на функції, що генеруються.

Для більшості сучасних задач з використанням булевих функцій можна виділити два основних етапи їх використання:

1. Визначення частини значень функції, виходячи з обмежень, що задаються криптографчним перетворенням;

2. Довизначення отриманої функції для решти значень так, аби вона задовольняла критеріям певним криптографічним критеріям, як-от максимуму повної та диференціальної ентропії.

При виконанні другого етапу виникає задача довизначення частково заданої булевої функції таким чином, аби вона була балансною та задовольняла критерію строгого лавинного ефекту.

Балансні функції з великою нелінійністю, що задовольняють критерію строгого лавинного ефекту, можуть бути отримані на основі bent-функцій [16], проте отримання самих цих функцій для великої кількості вхідних змінних є дуже

важкою задачею, вирішення якою потребує значних обчислювальних ресурсів з точки хору машинного часу та кількості необхідної пам’яті.

З іншої сторони, значно менших обчислювальних ресурсів потребує реалізація рекурсивного отримання балансних булевих SAC-функцій від n змінних з використанням чотирьох SAC-функцій від *n-1* змінних, які не обов’язково володіють критерієм балансності [1]. Однак, цей метод також не підходить для сформульованої задачі, оскільки на ці чотири функції також діє обмеження часткової їх визначеності.

Інший підхід [16] дозволяє отримувати лише незначну частину балансних SAC-функцій від загальної їх кількості. Він також не є доцільним для використання, оскільки на компоненти його вектору діють додаткові обмеження, які можуть бути виконані лише при виконанні перебору.

Таким чином, наявні методи створення балансних функції з властивістю строгого лавинного ефекту не підходять для реалізації визначеної у другому розділі задачі, а тому необхідно створити новий метод вирішення задачі довизначення булевої функції так, аби вона відповідала SAC.

3.2. Розробка методу довизначення булевих перетворень для забезпечення стійкості до диференційного криптоаналізу

При вирішення поставленої задачі вхідною є функція $f(x_1, \dots, x_n)$ від n змінних, що є визначеною на M наборах. Очевидно, що в залежності від цього значення M, задача довизначення функції f так, аби вона задовольняла строгому лавинному ефекту, може бути вирішена з певною вірогідністю. Причому чим більшим є це значення M до максимальної кількості наборів, на яких функція визначена (2^n), тим вірогідність успішного довизначення функції для SAC є меншою.

Виходячи з цього, головну мету поставленої задачі можна сформулювати таким чином: довизначити вхідну функцію так, аби вона в найбільшій мірі задовольняла SAC. Додатковою задачею є дослідження залежності виконання строгого лавинного ефекту від наперед заданого значення *M*.

Запропонований далі метод буде базуватися на тому факті, що якщо функція $f(x_1, \dots, x_n)$ від n змінних та її часткові похідні $\frac{\partial f}{\partial x_1} \dots \frac{\partial f}{\partial x_n}$ є балансними, тобто якщо кількість одиниць та нулів у таблицях істинності цих функцій є однаковим, то з цього випливає, що функція $f(x_1, \dots, x_n)$ задовольняє строгому лавинному ефекту.

Таким чином, вхідними даними для поставленої задачі довизначення функції є заповнена для M наборах таблиця істинності функції, яку потрібно довизначити.

Головну ідею запропонованого методу можна сформулювати таким чином. Метод полягає у послідовному дозаповненню таблиці істинності функції $f(x_1, \dots, x_n)$ так, аби якнайкращим чином збалансувати кількість наборів, для яких функція приймає нульове та одиничне значення, причому потрібно, аби те ж саме виконувалося як для самої функції, так і для її часткових похідних $\frac{\partial f}{\partial x_1} \dots \frac{\partial f}{\partial x_n}$.

Запропонований метод довизначення функції з метою забезпечення її лавинних властивостей зводиться до виконання такої послідовності кроків:

1. За допомогою відомих (визначених) значень функції $f(x_1, \dots, x_n)$ визначити значення її часткових похідних $\frac{\partial f}{\partial x_1} \dots \frac{\partial f}{\partial x_n}$. Для визначення часткової похідною $\frac{\partial f}{\partial x_i} = g(x_1, \dots x_{i-1}, x_{i+1} \dots, x_n)$ по змінній x_i, де $i \in \{1..n\}$, аналізуються усі 2^{n-1} набори змінних $x_1, \dots x_{i-1}, x_{i+1} \dots, x_n$. Кожному з цих наборів відповідає два набори усіх n змінних: $x_1, \dots x_{i-1}, 0, x_{i+1} \dots, x_n$ та $x_1, \dots x_{i-1}, 1, x_{i+1} \dots, x_n$. Якщо під час обрахунку часткової похідної функція $f(x_1, \dots, x_n)$ є визначеною на обох наборах та має одне і те саме значення, то значення похідної приймається рівним нулю для цих двох наборів, в інакшому випадку, тобто коли функція $f(x_1, \dots, x_n)$ приймає різні значення, то відповідні два значення похідної приймаються рівними

одиниці. У випадку, коли функція $f(x_1, \dots, x_n)$ є визначеною лише на одному з двох наборів, відповідні значення часткової похідної визначаються як частково визначені значення. Для позначення частково визначених значень використовується символ «?». Якщо ж функція $f(x_1, \dots, x_n)$ не є визначеною на обох наборах, то відповідні значення часткової похідної на обох наборах приймається невизначеним і позначається через «-».

2. Для оцінки можливих варіантів значень функції $f(x_1, \dots, x_n)$ кожне її значення, що не є визначеним, замінюється обома значеннями (нульовим та одиничним), і кожне з цих значень приймається до розгляду. Після цього обраховується оцінка впливу кожної з таких замін на властивість збалансованості як самої функції, так і її часткових похідних.

3. Функція довизначається відповідним значенням на тому наборі, інтегральна оцінка для якого має максимальне значення.

4. Проводиться корекція значення функції та її часткових похідних з урахуванням довизначення функції, що відбулося у третьому пункті для обраного набору. Якщо на цьому етапі функція $f(x_1, \dots, x_n)$ все ще має набори, на яких вона не визначена, то виконується перехід до другого кроку, інакше роботу методу закінчено.

Для ілюстрації роботи запропонованого методу далі наведено та детально прокоментовано роботу методу для конкретного прикладу.

Нехай є булева функція $f(x_1, x_2, x_3, x_4)$ від чотирьох змінних, що є частково визначеною на *M*=8 наборах. Таблиця істинності цієї функції наведена у таблиці 3.1, де через «-» позначено набори, на яких ця функція не є визначеною.

Згідно описаного методу, поступово виконується довизначення частково-визначеної булевої функції так, аби вона задовольняла критерію строгого лавинного ефекту.

Виконання першого етапу

Виконується обчислення похідних згідно заданих значень функції $f(x_1, x_2, x_3, x_4)$. Як зазначалося вище, відповідне значення часткової похідної приймається рівним нулю, якщо відповідні значення функції $f(x_1, x_2, x_3, x_4)$ на обох наборах (при інвертації значення змінної, по якій виконується диференціювання) є рівними, в інакшому випадку значення часткової похідної приймається рівним одиниці.

Таблиця 3.1

Таблиця істинності функції f

x_1, x_2, x_3, x_4	f	x_1, x_2, x_3, x_4	f
0000	0	1000	0
0001	-	1001	0
0010	0	1010	-
0011	1	1011	-
0100	-	1100	1
0101	-	1101	-
0110	-	1110	1
0111	0	1111	-

Значення часткових похідних функції $f(x_1, x_2, x_3, x_4)$ наведені у таблиці 3.2. Частково визначені значення похідних позначаються як «?», а невизначені – як «-».

Таблиця 3.2

Таблиця істинності часткових похідних та самої функції f

x_1, x_2, x_3, x_4	f	$\frac{\partial f}{\partial x_1}$	$\frac{\partial f}{\partial x_2}$	$\frac{\partial f}{\partial x_3}$	$\frac{\partial f}{\partial x_4}$	x_1, x_2, x_3, x_4	f	$\frac{\partial f}{\partial x_1}$	$\frac{\partial f}{\partial x_2}$	$\frac{\partial f}{\partial x_3}$	$\frac{\partial f}{\partial x_4}$
0000	0	0	?	-	?	1000	0	0	1	?	0
0001	-	?	-	?	?	1001	0	?	?	?	0
0010	0	?	?	0	1	1010	-	?	?	?	-
0011	1	?	1	?	1	1011	-	?	-	?	-
0100	-	?	?	-	-	1100	1	?	?	0	?
0101	-	-	-	?	-	1101	-	-	-	-	?
0110	-	?	?	-	?	1110	1	?	?	0	-
0111	0	?	1	?	?	1111	-	?	1	-	-

Виконання другого етапу

Послідовно, замість кожного не визначеного значення функції, підставляється нульове та одиничне пробне значення. Після цього обчислюються часткові похідні, значення яких змінилися в результаті цієї підстановки. Значення похідних, які змінилися у результаті цієї операції, наведені у таблиці 3.3.

Після цього оцінюються результати проведеної заміни. Для цього підраховується кількість одиниць та нулів функції та її часткових похідних. У функції f визначено три одиниці та п’ять нулів, тобто на поточному кроці при виборі наступного значення для функції необхідно віддати перевагу одиниці.

Таблиця 3.3

Таблиця пробних значень функції f та її часткових похідних

x_1, x_2, x_3, x_4	f	$\frac{\partial \mathrm{f}}{\partial \mathrm{x}_1}$	$\frac{\partial \mathrm{f}}{\partial \mathrm{x}_2}$	$\frac{\partial \mathrm{f}}{\partial \mathrm{x}_3}$	$\frac{\partial \mathrm{f}}{\partial \mathrm{x}_4}$	x_1, x_2, x_3, x_4	f	$\frac{\partial \mathrm{f}}{\partial \mathrm{x}_1}$	$\frac{\partial \mathrm{f}}{\partial \mathrm{x}_2}$	$\frac{\partial \mathrm{f}}{\partial \mathrm{x}_3}$	$\frac{\partial \mathrm{f}}{\partial \mathrm{x}_4}$
0001	1	1	?	0	1	1010	1	1	0	1	?
	0	0	?	1	0		0	0	1	0	?
0100	1	0	1	?	?	1011	1	0	?	1	?
	0	1	0	?	?		0	1	?	0	?
0101	1	?	?	1	?	1101	1	?	1	?	0
	0	?	?	0	?		0	?	0	?	1
0110	1	0	1	?	1	1111	1	1	?	?	0
	0	1	0	?	0		0	0	?	?	1

Для похідної $\frac{\partial f}{\partial x_1}$ кількість нулів рівна двум, а кількість одиниці рівна нулю. Значить, потрібно віддати перевагу одиничному значенню. Аналогічно визначається, що для похідної $\frac{\partial f}{\partial x_2}$ потрібно вибрати нульове значення, для $\frac{\partial f}{\partial x_3}$ – одиничне значення, а для $\frac{\partial f}{\partial x_4}$ будь-яке значення є підходящим, оскільки кількість одиниць рівна кількості нулів.

У результаті наведених вище підрахунків, формується таблиця 3.4, у які наведено оцінка пробних значень функції та її похідних. У цій таблиці через символ «+» позначено підходящі значення, через символ «−» позначено непідходящі значення, а через символ «±» позначено допустимі значення.

Таблиця 3.4

Таблиця оцінки пробних значень функції *f* та її часткових похідних

x_1, x_2, x_3, x_4	f	$\frac{\partial f}{\partial x_1}$	$\frac{\partial f}{\partial x_2}$	$\frac{\partial f}{\partial x_3}$	$\frac{\partial f}{\partial x_4}$	x_1, x_2, x_3, x_4	f	$\frac{\partial f}{\partial x_1}$	$\frac{\partial f}{\partial x_2}$	$\frac{\partial f}{\partial x_3}$	$\frac{\partial f}{\partial x_4}$
0001	+	+	±	−	±	1010	+	+	+	+	±
	−	−	±	+	±		−	−	−	−	±
0100	+	−	−	±	±	1011	+	−	±	+	±
	−	+	+	±	±		−	+	±	−	±
0101	+	±	±	+	±	1101	+	±	−	±	±
	−	±	±	−	±		−	±	+	±	±
0110	+	−	−	±	±	1111	+	+	±	±	±
	−	+	+	±	±		−	−	±	±	±

Виконання третього етапу

Аналізуючи значення у таблиці 3.4, визначається набір, для якого кількість підходящих значень є найбільшою. Кількість «+» для першої строчки рівно двум, для другого – одиниця, і так далі. Найбільша кількість підходящих значень знаходиться у десятій строчці, для варіанту *f(1010)=1*, а значить вибране значення функції *f* записується до таблиці істинності і вважається визначеним набором.

Виконання четвертого етапу

Виконується корекція значень функції та її похідних з урахуванням визначеного значення функції на вибраному наборі.

Перераховані вище кроки виконуються до тих пір, поки функція *f* не стане визначеною для всіх наборів. Результатом виконання наведеного прикладу є таблиця істинності для функції *f*, яка тепер задовольняє SAC, що наведена у таблиці 3.5.

Таблиця 3.5

Таблиця істинності функції *f* після виконання методу

x_1, x_2, x_3, x_4	f	x_1, x_2, x_3, x_4	f
0000	0	1000	0
0001	1	1001	0
0010	0	1010	1
0011	1	1011	1
0100	1	1100	1
0101	0	1101	0
0110	0	1110	1
0111	0	1111	1

Таким чином, задача довизначення частково-визначеної булевої функції так, аби вона відповідала строгому лавинному ефекту, є вирішеною. Описаний метод може бути використана для довизначення сформованих таблиць функціональних перетворювачів, які генеруються згідно описаного у другому розділі методу генерації хеш-перетворення на основі булевого функціонального перетворювача, аби отримане хеш-перетворення якнайкраще задовольняло строгому лавинному ефекту для досягнення ефективного рівня криптографічного захисту.

Висновки до розділу 3

По проведеній у третьому розділі роботі можна зробити такі висновки:

1. Дано визначення та описано важливість таких критеріїв булевих функцій, як балансність та строгий лавинний ефект (SAC). Показано, що для забезпечення високого рівня криптографічного захисту необхідно, аби булева функція задовольняла цим критеріям.

2. Запропоновано метод довизначення частково-визначеної булевої функції так, аби отримана функція задовольняла строгому лавинному ефекту. Наведено детальний приклад роботи запропонованого методу.

3. Описаний метод довизначення булевої функції може бути використаний для завершальної частини методу генерації хеш-перетворення, який було запропоновано у другому розділі, для того аби підвищити криптографічну стійкість отриманого булевого функціонального перетворювача за рахунок виконання критерію строгого лавинного ефекту.

РОЗДІЛ 4

РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ ПОБУДОВИ БУЛЕВИХ ПЕРЕТВОРЕНЬ ДЛЯ ІДЕНТИФІКАЦІЇ ТА АПАРАТНИХ ЗАСОБІВ ЇЇ ШВИДКОЇ РЕАЛІЗАЦІЇ

Для можливості надання квантитативної оцінки запропонованому у другому розділі алгоритму генерації хеш-перетворення необхідно перевірити його роботу для різної множини вхідних даних та кількісно оцінити його вихідні дані.

Завдання алгоритму можна сформулювати наступним чином: для заданої множини вхідних параметрів заповнити таблиці функціональних перетворювачів таким чином, аби отримати явним чином множину вхідних паролів користувача, таку, що застосування згенерованого булевого функціонального перетворення до кожного з них дає на виході один і той самий ідентифікатор користувача.

Виходячи із такого опису завдання алгоритму, до програмного застосунку для його тестування доцільно висунути наступні вимоги:

- він має надавати можливість встановлення значень вхідних параметрів роботи алгоритму, таких як кількості m фрагментів розбиття ідентифікатору користувача, бітової довжини k кожного з фрагментів, а також допоміжні налаштування для перевірки правильності роботи алгоритму;
- після завершення роботи алгоритму програмний застосунок має виводити на екран кількість паролів, які вдалося згенерувати у рамках заданих параметрів роботи;
- для можливості перевірки правильності роботи алгоритму вручну, програмний застосунок має мати змогу виведення на екран множини згенерованих паролів та ідентифікатора користувача, для якого проводилася генерація паролів;

- програмний застосунок має містити механізм підтвердження правильності роботи алгоритму у вигляді перевірки кожного зі згенерованих паролів наступним чином: при застосування до нього кінцевого булевого функціонального перетворення, на значення виході рівне ідентифікатору користувача, для якого генерується множина паролів;
- програмний застосунок має мати змогу робити правильно та швидко навіть для чисел з великою розрядністю.

У якості мови програмування для створення програмного застосунку був обраний Rust. Такий вибір можна обґрунтувати наступним чином:

- мова Rust є сучасною системною мовою програмування, яка поєдную в собі швидкість роботи низько-рівневої мови (Rust можна прирівняти по швидкості до C та C++) та зручність використання високо-рівневої мови (окрім сучасної бібліотеки стандартних інструментів, Rust також підтримує парадигми функціонального програмування);
- особливістю мови програмування Rust є його механізм строгої перевірки типів та контролювання володіння об’єктами на момент компіляції, що призводить до того що успішна компіляція у більшості випадків означає правильну роботу програми, за умови відсутності логічних помилок;
- Rust є популярною та сучасною мовою програмування, яку люблять та поважають. Вона має якісну та чітку офіційну документацію, а також безліч інших користувачів, які завжди готові дати відповідь на будь-які запитання.

Проекти у мові програмування Rust створюються за допомогою утиліти cargo, яка створює директорію проекту разом з другорядними файлами, такими як *Cargo.toml* для встановлення налаштувань проекту та його залежностей. Для компіляції та запуску програми доступні команди *cargo check* для швидкої компіляції без генерації бінарника, а також *cargo run* для компіляції з генерацією

готового бінарника. Для створення оптимізованого і швидкого бінарника використовується команда *cargo run --release*.

На сьогоднішній день компілятори стали набагато кращими у питаннях оптимізації коду: чим більше конкретної інформації доступно на момент компіляції, тим краще компілятори можуть оптимізувати код та пришвидшити виконання програми. Оскільки одним із ключових потреб, поставлених до програмного застосунку, є швидкість роботи, то параметри для алгоритму встановлюються безпосередньо у сирцевому коді, що робить їх відомими ще на етапі компіляції та дозволяє якнайкраще розподілити пам’ять та оптимізувати програмний код.

Далі буде наведено опис таких глобальних параметрів, за допомогою яких керується робота програмного застосунку. Вони розміщені на початку сирцевого файлу *main.rs*.

4.1. Структура даних програми

[type; count]

Є інструментом оголошення масиву довільного типу даних конкретної довжини, що має бути константою на момент компіляції. Оголошується за допомогою квадратних дужок, у яких першим аргументом вказується тип для зберігання, а другим — кількість елементів. Два аргументи розділяються крапкою з копою.

Якщо потрібно створити новий масив, то це робиться так:

let my_array = [u32; 10];

Такий рядок створить масив на 10 елементів, кожен з яких є 32-бітним цілим числом без знаку.

Перевагою такого масиву є максимальна швидкість доступу та зміни окремих елементів. Недоліком є відсутність можливості зміни довжини, тому

використовується він тоді, коли кількість елементів не змінюється на протязі його існування.

Vec<T>

Дозволяє створити вектор (масив змінюваної довжини), що зберігатиме елементи типу *T*. “Під капотом” реалізується за допомогою звичайного масиву, проте надає зручний інтерфейс, що дозволяє не займатися операціями по зміні довжині самостійно. Має таку саму швидкість доступу та зміни окремих елементів, проте потребує часу, коли змінюється його внутрішня довжина. Окрім можливості додавати та видаляти елементи, вектор також має безліч зручних функцій для роботи з елементами.

Для того щоб оголосити новий вектор потрібно написати:

let my_vec = Vec::new();

Тип елементів буде визначено автоматично, коли вектор буде використовуватися. Альтернативним варіантом створення вектору є використання макросу:

let my_vec2 = vec![1,2,3];

Цей метод зручний, коли заздалегідь відомий початковий вміст та розмір вектору.

Якщо відома максимальна довжина вектору, який буде використовуватися, то можна отримати найкраще з обох світів: звичайного масову так вектору. Для цього можна використати метод

let me_vec_3 = Vec::with_capacity(10);

який один раз виділить потрібний об’єм пам’яті, після чого їм можна користуватися як і будь-яким іншим вектором без втрат ефективності, адже пропадають витрати на зміну внутрішнього розміру вектору.

```
struct Number {
    value: usize,
}
```

Структура *Number* є основним типом чисел, які використовуються під час роботи алгоритму. Саме цей тип буде використовуватися для зберігання вмісту комірок, для зберігання вхідного та вихідного векторів, а також для проведення усіх внутрішніх операції над ним.

Єдиним внутрішнім полем структури є *value*, що має тип *usize*. Тип *usize* у кожній системі дорівнює типу, що може зберегти вказівник у цій архітектурі системи. Для 64-бітних архітектур *usize* має такий самий розмір, як і *u64*. Окремий тип використовується для того, аби відрізняти за смислом звичайні числа та ітератори і довжину об’єктів.

Для роботи зі структурою *Number* було створено такі асоційовані функції:

- *fn empty() -> Number* – функція, яка створює новий екземпляр *Number*, що містить порожнє значення, яке репрезентується за допомогою числа *EMPTY_NUMBER*. Ця функція є аналогом конструктора за замовчуванням з об’єктно-орієнтованої мови програмування. Викликається від імені типу *Number*.

- *fn new(value: usize) -> Number* – функція, яка створює новий екземпляр *Number*, що містить значення, рівне переданому в аргументи *value*. Ця функція є аналогом звичайного конструктора з об’єктно-орієнтованої мови програмування. Викликається від імені типу *Number*.

- *fn empty(&self) -> bool* – функція, яка перевіряє вміст *Number*, для якого вона викликається, та повертає значення *true*, якщо це число відповідає порожньому числу (його внутрішнє значення value рівне *EMPTY_NUMBER*), інакше повертається *false*. Викликається від імені об’єкта з типом *Number*.

- *fn xor(&self, other: &Number) -> Number* – функція, яка проводить булеву побітову операцію виключного АБО над цим числом та числом, що було передано

в аргумент функції. Результат виконання обертається в новий об’єкт *Number* та повертається з функції. Викликається від імені об’єкта з типом *Number*.

Для виводу значення *Number* на екран було прописано реалізацію *Display trait* для типу *Number*, в якій задається функція *fn fmt(&self, &mut std::fmt::Formatter) -> std::fmt::Result*.

Для того аби не прописувати один і той самий тип багато разів, у мові програмування *Rust* можна задати аліас для типу. Такий аліас створено для масивів вхідного та вихідного векторів:

t*ype FragmentsOfX = [Number; XS_COUNT];*

type FragmentsOfY = [Number; YS_COUNT];

Для роботи з вхідними та вихідними векторами створено декілька допоміжних функцій, які використовують новостворені аліаси:

- *fn equal(a: &FragmentsOfX, b: &FragmentsOfX) -> bool* – функція, яка перевіряє поелементну рівність двох вхідних векторів (паролів користувача). Завдяки зручному інтерфейсу ітераторів у мові програмування *Rust* ця функція має дуже просту реалізацію.

- *fn not_equal(a: &FragmentsOfX, b: &FragmentsOfX) -> bool* – функція, яка перевіряє поелементну рівність двох вхідних векторів (паролів користувача), та повертає значення *true*, якщо масиви відрізняються хоча б в одному елементі, інакше повертає *false*. Завдяки зручному інтерфейсу ітераторів у мові програмування *Rust* ця функція має дуже просту реалізацію.

- *fn not_equal_array_y(a: &FragmentsOfY, b: &FragmentsOfY)* – функція, яка перевіряє поелементну рівність двох вихідних векторів (ідентифікаторів користувача), та повертає значення *true*, якщо масиви відрізняються хоча б в одному елементі, інакше повертає *false*. Завдяки зручному інтерфейсу ітераторів у мові програмування *Rust* ця функція має дуже просту реалізацію.

- *fn unique(t: &FragmentsOfX, ts: &Vec<FragmentsOfX>) -> bool* – функція, яка перевіряє, що перший аргумент (пароль користувача) не зустрічається у переданому у якості другого аргументу масиві (множині паролів користувача). Можна сказати, що ця функція перевіряє унікальність нового паролю користувача. Завдяки зручному інтерфейсу ітераторів у мові програмування *Rust* ця функція має дуже просту реалізацію.
- *fn print_array_x(x: &FragmentsOfX)* – функція, яка виконує вивід на екран переданого в аргументи паролю користувача, виводячи послідовно кожний елемент та додаткові символи для зручності.
- *fn print_array_y(x: &FragmentsOfY)* – функція, яка виконує вивід на екран переданого в аргументи ідентифікатору користувача, виводячи послідовно кожний елемент та додаткові символи для зручності.

```
struct Func {
    map: Vec<Number>,
    cache: Vec<Vec<Number>>,
    empty_count: usize,
}
```

Ця структура відповідає за збереження необхідних для роботи функціонального перетворювача даних. Можна сказати, що вона є його аналогом з точки зору програми.

Перше поле структури — *map* — зберігає безпосередньо таблицю з комірками функціонального перетворювача. Для кожного індексу, що відповідає адресі комірки, зберігається об’єкт типу *Number*, що відповідає значенню комірки.

Друге поле структури — *cache* — є оптимізацією роботи функціонального перетворювача, що дозволяє значно пришвидшити виконання алгоритму генерації хеш-перетворення, витративши додаткову пам’ять на зберігання кешу. Це поле

зберігає множину адрес комірок, кожна з яких містить значення, яке співпадає з індексом у масиві, де лежить ця множина. По суті, цей кеш є швидким механізмом для отримання множини комірок, що містять одне й те саме необхідне значення.

Третє поле структури — *empty_count* — зберігає поточну кількість порожніх комірок у всій таблиці функціонального перетворювача. Зберігання та своєчасна зміна цього значення дозволяє не виконувати пошук по величезній таблиці і максимально зменшити час отримання цього значення.

Кожне з вищеописаних полів структури змінюється належним чином під час використання асоційованих функції структури *Func*. Асоційованими функціями для цієї структури є такі:

- *fn new() -> Func* – функція, яка створює новий об’єкт типу *Func* і ініціалізує його поля відповідним чином. А саме, поле *map* ініціалізується новим вектором, що одразу отримує довжину, рівну кількості елементів, що будуть там зберігатися — *NUMBERS_COUNT*. Це дозволяє виконувати подальші операції по доступу до вектору, якби це був звичайний масив, але при цьому мати доступ до усіх зручних функцій вектору. Кожне з чисел цього вектору встановлюється у порожнє значення, бо створюється викликом метода *Number::empty()*. Аналогічно, поле *cache* також ініціалізується як вектор розміру *NUMBERS_COUNT*, проте його елементами є пусті вектори. Останнє поле *empty_count* ініціалізується значенням *NUMBERS_COUNT*, оскільки саме стільки порожніх комірок містить функціональний перетворювач на початку роботи. Викликається від імені типу *Func*.

- *fn set(&mut self, arg: Number, value: Number)* – функція, яка змінює для функціонального перетворювача, для якого вона викликається, вміст комірки за адресою *arg* на *value*. При цьому в залежності від того, чи є старий та новий вміст комірки порожніми значеннями, буде змінено значення поля *empty_count* належним чином, аби це поле завжди зберігало правильну поточну загальну кількість

порожніх комірок. Також буде змінено вміст кешу *cache*, оскільки з нього потрібно видалити інформацію по індексу, що рівний старому вмісту комірки, та додати інформацію по індексу, що рівний новому вмісту комірки. Таким чином, значення кешу залишається коректним. Викликається від імені об’єкта з типом *Func*.

- *fn at(&self, arg: usize) -> Number* – функція, яка повертає поточне значення з таблиці функціонального перетворювача, для якого вона викликається, за адресою arg. Викликається від імені об’єкта з типом *Func*.

- *fn empty_at(&self, arg: Number) -> bool* – функція, яка перевіряє, чи є поточне значення таблиці функціонального перетворювача, для якого вона викликається, за адресою arg порожнім. Якщо ця комірка містить порожнє значення, то функція поверне значення *true*, інакше — *false*. Викликається від імені об’єкта з типом *Func*.

- *fn get_empty_count(&self) -> usize* – функція, яка поверне амортизоване значення кількості порожніх на даний момент комірок у таблиці функціонального перетворювача, для якого викликається ця функція. Завдяки виконанню такої оптимізації, а саме підрахунку порожніх комірок на кожному етапі зміни таблиці, виклик цієї функції є дуже швидким. Викликається від імені об’єкта з типом *Func*.

Для того аби мати можливість вивести на екран поточний вміст функціональних перетворювачів, була створена спеціальна функція:

fn print_funcs(funcs: &Vec<Func>)

яка зручним чином виводить таблиці множини усіх функціональних перетворювачів. Для легкості перевірки правильності роботи алгоритму, ця функція виводить у кожному рядку відповідне значення комірки по одному і тому самому індексу усіх функціональних перетворювачів, після чого переходить до наступної адреси комірки (індексу) і виводить відповідні їх значення уже на наступному рядку.

4.2. Розробка функціональних модулів програми

Однією з найважливіших функцій усього алгоритму є *fn run_through(funcs: &Vec<Func>, start_x: &FragmentsOfX) -> FragmentsOfY*, яка для заданої множини функціональних перетворювачів із заповненими таблицями, що передається першим аргументом, бере вхідний вектор (пароль користувача), що передається другим аргументом, та пропускає цей вектор через функціональні перетворювачі, згідно архітектури булевого функціонального перетворювача. Фактично, виконує головний етап ідентифікацію користувача. Після виконання усіх перетворень функція отримує та повертає значення *FragmentsOfY*, що відповідає ідентифікатору користувача, який генерує хеш-перетворення з переданим паролем користувача *FragmentsOfX*.

Зважаючи на те, що функція run_through виконує основне перетворення, що буде виконуватися при кожній ідентифікації користувача у системі, то доречно розглянути вихідний код реалізації цієї функції детальніше:

```
fn run_through(funcs: &Vec<Func>, start_x: &FragmentsOfX) -> FragmentsOfY
{
    let mut x = start_x.clone();
    for width in ((YS_COUNT + 1)..=XS_COUNT).rev() {
        for j in 0..width {
            x[j] = funcs[j].at(x[j].value);
        }
        for j in 0..(width - 1) {
            x[j] = x[j].xor(&x[j + 1]);
        }
    }
    for j in 0..FRAGMENTS_COUNT {
        x[j] = x[j].xor(&start_x[j]);
```

```
    }
    x_to_y(x)
}
```

Першим кроком функція робить повну копію паролю користувача, а саме значення типу *FragmentsOfX*, яке необхідно пропустити через хеш-перетворення. Це робиться тому, що далі саме над цим масивом будуть проводитися усі математичні операції, і саме в ньому буде зберігатися поточний стан поточного шару перетворення. Оскільки довжина цього масиву є невеликою (декілька десятків елементів), що операцією копіювання можна знехтувати.

Наступним кроком функція виконує цикл, який для кожного шару булевого функціонального перетворювача, якому можна співставити певне значення ширини цього шару (їй відповідає змінна *width* у коді), виконує прохід, підставляючи поточні значення у якості вхідних значень до функціональних перетворювачів, отримуючи значення за цими адресами з таблиць функціональних перетворювачів, а потім, отримавши масив значень на виході з функціональних перетворювачів, виконує згортки для наступного шару, згідно архітектури, за допомогою операції виключного АБО. Наступний шар, як наслідок, буде мати ширину, що на один менша за поточну.

Останнім кроком є виконання тієї самої операції згортки, але з використанням вхідного вектору (паролю користувача). Оскільки на початку функції була виконана копія цього вектору, то його значення ще зберіглося і може бути використане.

Після виконання описаних перетворень, у поточному масиві на перших *YS_COUNT* елементах зберігаються вихідні значення ідентифікатору користувача, які вилучаються викликом функції *x_to_y*. Отримане значення вектору є значенням, що повертається з функції.

Для перевірки коректності паролів користувача, що були створені за допомогою алгоритму, використовується функція

fn hashed_match(funcs: &Vec<Func>, xs: &Vec<FragmentsOfX>, y: &FragmentsOfY) -> bool

яка для заданої множини функціональних перетворювачів із заповненими таблицями, що передається першим аргументом, почергово бере кожен із паролів користувача із їх множини, що передається другим аргументом, та перевіряє, що отримане після застосування булевого функціонального перетворення до поточного паролю користувача значення отриманого ідентифікатору користувача співпадає з тим, що передається третім аргументом до функції. Якщо усі паролі користувача виявилися коректними, то функція повертає значення *true*, а інакше, якщо хоча б один пароль дів неправильне значення ідентифікатору користувача, то повертається *false*.

Отримання псевдовипадкових чисел

Для генерації псевдовипадкових чисел з рівномірним законом розподілу у використовується бібліотека *rand*, що має бути підключена у *Cargo.toml*. На момент створення програмного застосунку використовувалась версія 0.8.3 цієї бібліотеки, яку правильно називати *crate* у середовищі *Rust*.

Для використання псевдовипадкового генератора чисел за допомогою цієї бібліотеки необхідно створити новий об’єкт за допомогою виклику *rand::thread_rng()*. Далі цей об’єкт *random* буде передаватися в усі функції, які так чи інакше потребують генерації випадкових чисел.

Для зручності роботи з цим об’єктом були створені такі додаткові функції:

- *fn generate_random_uniform_int(random: &mut rand::rngs::ThreadRng, low: usize, high: usize) -> usize* – функція, яка на основі двох аргументів, що задають нижню та верхню границю для діапазону, викликає функцію *gen_range* для переданого об’єкту *random*, передаючи їх створений діапазон у якості параметру.

На виході генеруються псевдовипадкове число у вказаному діапазону з рівномірним законом розподілу, це значення і повертається із функції.

- *fn generate_random_number(random: &mut rand::rngs::ThreadRng) -> Number* – функція, яка створює новий об’єкт типу Number, ініціалізуючи його з рівною ймовірністю будь-яким з коректних для нього значень з рівною ймовірністю кожного значення. Коректними значеннями є усі числа з діапазону від нуля до *MAX_NUMBER*. Генерація виконується за допомогою виклику функції *generate_random_uniform_int*, якій передається вказана нижня та верхня границя у якості параметрів.

- *fn generate_random_array_y(random: &mut rand::rngs::ThreadRng) -> FragmentsOfY* – функція, яка генерує новий випадковий ідентифікатор користувача *FragmentsOfY*, ініціалізуючи кожний елемент цього масиву випадковим значенням, отриманим за допомогою функції *generate_random_number*.

Допоміжні функції для роботи основного алгоритму

При роботі основного алгоритму виникає необхідність багаторазово повторювати одні і ті ж операції, тому доцільно винести їх у окрему функцію для наочності та дати їм зрозуміле ім’я.

Такими функціями є:

- *fn pick_random_empty(random: &mut rand::rngs::ThreadRng, func: &Func, forbidden_index: usize) -> Number* – функція, яка отримує на вхід функціональний перетворювач, та повертає адресу випадкової порожньої комірки у ньому, причому її адреса не може бути рівною *forbidden_index*, що передається останнім аргументом. Якщо не існує такої комірки, що відповідає поставленим до неї вимогам, то метод повертає *Number::empty()*. Для того аби кожна з порожніх комірок мала рівний шанс бути вибраною, функція починає пошук комірки з випадково згенерованої адреси початкової комірки, після чого послідовно шукає

першу комірку у таблиці функціонального перетворювача, що відповідає поставленим вимогам. Дійшовши до кінця таблиці, пошук продовжується з першої комірки. Цикл гарантовано знайде необхідну комірку, адже її необхідна наявність перевіряється перед початком циклу.

- *fn pick_random_not_empty(random: &mut rand::rngs::ThreadRng, func: &Func) -> Number* – функція, яка отримує на вхід функціональний перетворювач, та повертає адресу випадкової не порожньої комірки у ньому. Якщо не існує такої не порожньої комірки, то метод повертає *Number::empty()*. Для того аби кожна з не порожніх комірок мала рівний шанс бути вибраною, функція починає пошук комірки з випадково згенерованої адреси початкової комірки, після чого послідовно шукає першу комірку у таблиці функціонального перетворювача, що не є порожньою. Дійшовши до кінця таблиці, пошук продовжується з першої комірки. Цикл гарантовано знайде необхідну комірку, адже її необхідна наявність перевіряється перед початком циклу.

- *fn pick_random_target(random: &mut rand::rngs::ThreadRng, func: &Func, target: usize) -> Number* — функція, яка отримує на вхід функціональний перетворювач, та повертає адресу випадкової комірки у ньому, що містить передане останнім аргументом значення *target*. Якщо не існує такої комірки у переданому функціональному перетворювачі, функція повертає значення *Number::empty()*. Ця функція виконується швидко, адже використовує для своїх роботи поле *cache* об’єкту *Func*, виконуючи пошук серед комірок, що зберігаються у кеші для вказаного значення *target*.

Реалізація основного алгоритму

Усі описані до цього структури даних та функції є допоміжними, і будуть використовуватися у функції *main*, на яку покладена реалізація основного алгоритму.

Глобально, роботу функції main можна описати наступним чином. По-перше, згенерувати ідентифікатор користувача за допомогою генератора псевдовипадкових чисел. По-друге, продовжувати генерувати паролі користувача та заповнювати при цьому таблиці функціональних перетворювачів, допоки виконується основна умова алгоритму і функціональні перетворювачі містять достатню кількість порожніх комірок перед початком виконання чергової ітерації. По-третє, за необхідністю, перевірити на правильність усі згенеровані паролі та вивести результати роботи алгоритму на екран (кількість паролів, яку вдалося згенерувати).

Далі буде детально розглянуто послідовність кроків програмної реалізації алгоритму.

Спочатку програма виводить повідомлення у термінал про налаштування, з якими була запущена програма, а саме значення змінних *FRAGMENTS_BITNESS* та *FRAGMENTS_COUNT*, які показують бітову довжину одного фрагмента та загальну кількість фрагментів для вихідного вектора.

Наступним кроком є створення масиву функціональних перетворювачів, з яких складається булевий функціональний перетворювач. Їх буде *FUNCS_COUNT* штук, кожен з яких створюється викликом функції *Func::new()*.

Далі необхідно згенерувати випадковий ідентифікатор користувача згідно правил алгоритму. Для цього використовується заздалегідь написана функція, а результат зберігається до змінної *y*:

let y = generate_random_array_y()

Якщо для програми налаштування *COMMENT* встановлено у значення *true*, то на цьому кроці виводиться згенероване значення вихідного вектору за допомогою функції *print_array_y*, щоб можна було вручну перевірити правильність алгоритму.

Після цього необхідно підготувати вектор, до якого будуть зберігатися згенеровані паролі користувача. Створюється нова змінна *xs*, і ініціалізується новим пустим вектором.

Ключовою перевіркою, згідно якої алгоритм приймає рішення, чи можна продовжувати генерацію паролів, є перевірка на те, що кожен з функціональних перетворювачів містить принаймні *(FRAGMENTS_COUNT - 1)* порожніх комірок. Якщо ця умова виконується, то можна продовжувати генерацію паролів. Перевірка цієї умови винесена у окрему лямбда-функцію, що завдяки мові *Rust* має дуже компактну реалізацію:

```
let all_funcs_sufficient = |funcs: &Vec<Func>| {
    funcs.iter().all(|f| f.get_empty_count() >= FRAGMENTS_COUNT - 1)
};
```

У ній створюється ітератор для вектору функціональних перетворювачів, після чого для нього викликається метод *all()*, який поверне результат *true*, якщо предикат, переданий йому в аргументи, буде істинним для кожного елементу вектору. У якості предикату передається лямбда-функція, яка, беручи в аргументи функціональний перетворювач, порівнює кількість порожніх комірок, що залишилися в ньому, зі значенням *(FRAGMENTS_COUNT - 1)*, і повертає *true*, якщо кількість порожніх комірок більша за це значення.

З метою підрахунку кількості паролів, що були за якоюсь причиною згенеровані повторно, вводиться змінна *duplicates_count*, що буде збільшуватися на 1 кожен раз у кінці ітерації, якщо черговий пароль виявиться не унікальним. При правильній роботі алгоритму таких ситуацій не повинно виникати, або їх кількість має бути близькою до 0.

Далі починається безпосередньо заповнення таблиць функціональних перетворювачів та генерація паролів.

Виконується основний цикл програми, на кожній ітерації якого буде створено новий пароль. Оскільки на початку роботи усі комірки усіх функціональних перетворювачів є порожніми, то умова *all_funcs_sufficient()* обов'язково виконується, а отже нема необхідності перевіряти її на початку роботи циклу. Натомість, ця умова перевіряється на кінці кожної ітерації.

Першим, що робить цикл, є перевірка налаштування *COMMENT*, і якщо вона встановлена у значення *true*, то користувач сповіщається про те, що на поточній ітерації усі функціональні перетворювачі містять достатню кількість порожніх комірок, а отже цикл продовжується.

Оголошується змінна i, яка міститиме індекс поточного шару, починаючи рахунок з 1. Її значення встановлюється у *(FRAGMENTS_COUNT - 1)*, що відповідає останньому шару.

Наступним кроком є оголошення масиву *v_last*, що складається з *(FRAGMENTS_COUNT + 1)* елементів, кожен з яких має значення *Number::empty()*. Цей масив зберігатиме значення на вході до функціональних перетворювачів останнього шару. Згідно алгоритму, усі значення масиву генеруються за допомогою функції *pick_random_empty()*, що викликається для кожного стовпчика. Оскільки на цьому етапі нема значень адрес комірок, які потрібно уникати, то останнім аргументом у якості *forbidden_index* передається значення *EMPTY_NUMBER*.

Наступним кроком є оголошення вектору *old_v*, що містить ту саму кількість елементів, що і масив *v_last*. Цей вектор міститиме значення на вході до функціональних перетворювачів попереднього шару по ходу виконання ітерації. Далі послідовно до цього масиву додаються усі елементи масиву *v_last*, оскільки у якості попереднього шару далі буде використовуватися останній. Не дивлячись на те, що *v_last* є вектором, пам'ять для його елементів виділяється один раз за допомогою методу *Vec::with_capacity()*, що дозволяє витрачати час та виділення пам'яті всього лише один раз.

Далі, згідно алгоритму, необхідно виконати обернений прохід з останнього шару (номер якого зберігається у змінній *i*) до першого шару булевого функціонального перетворювача. Тому починається цикл, що продовжує виконання, доки значення змінної *i* більше або дорівнює двом.

Першим кроком внутрішнього циклу необхідно перейти до попереднього шару, тому значення змінної *i* зменшується на 1.

Згідно п’ятого пункту алгоритму, необхідно обрати випадкову комірку за допомогою функції *pick_random_not_empty* у першому функціональному перетворювачі, що не є порожньою. При цьому, якщо такої комірки немає (тобто отримане з функції значення є пустим), то замість нього вибирається будь-яка комірка, що є порожньою, проте її значення не може співпадати зі значенням адреси першої комірки останнього шару, а саме *v_last[0]*. Тоді до функціонального перетворювача заноситься нове випадкове значення, згенероване випадковим чином, за отриманою адресою *v0*.

Обравши адресу *v0* тим чи іншим шляхом, згідно шостого пункту алгоритму необхідно обрахувати значення на вхід до функціональних перетворювачів наступного шару. Для цього оголошується вектор new_v, що містить щонайбільше *(2 * FRAGMENTS_COUNT - 1)* елементів. Єдиним відомим елементом цього вектору поки що є тільки обрахований v0, тому від додається до вектора.

Для усіх інших стовпчиків поточного шару розпочинається цикл, що іде від індексу 1 (оскільки нульовий стовпчик уже відомий — *v0*) до *(2 * FRAGMENTS_COUNT - i -1)*, що відповідає індексу передостаннього стовпчика на поточному шарі. Для кожного стовпчика виконується така послідовність дій.

За допомогою вхідного та вихідного значень з функціонального перетворювача попереднього стовпчику, обраховується значення на виході з поточного функціонального перетворювача *w* з використанням операції виключного АБО. Для того аби обчислити значення на вході поточного

функціонального перетворювача викликається функція *pick_random_target()*, якій у якості цільового значення комірки передається обраховане *w*. Якщо такої комірки немає, тобто якщо функція повернула порожнє число, то за допомогою функції *pick_random_empty()* знаходиться порожня комірка з адресою *v*, забороненим значенням для якої є або відповідне значення на вхід останнього шару (адже це значення є “зарезервованим” і не має використовуватися), або заборони на значення не існує зовсім, якщо індекс поточного стовпчика є більшим за кількість функціональних перетворювачів останнього шару. До функціонального перетворювача поточного стовпчику заноситься значення *w* за адресою *v*.

Отримавши значення для *v* тим чи іншим шляхом, воно додається до *new_v*, адже є черговим значенням на вході до функціонального перетворювача на поточному шарі.

Завершивши такий прохід по усім стовпчикам поточного шару, програма присвоює змінній *old_v* значення *new_v*, і переходить до наступного шару, якщо цей шар не був першим.

Коли було завершено обернений прохід до першого шару та згенеровано набір значень елементів, що їдуть на вхід до першого шару булевого функціонального перетворювача, то на цьому етапі стали відомі значення фрагментів вхідного вектора, адже вони поелементно рівні значенням на вхід останнього шару. Тому оголошується новий масив *x* довжини *XS_COUNT*, елементи якого ініціалізуються відповідними елементами з вектору *old_v*.

Далі алгоритм повертається до останнього шару. Випадковим чином за допомогою функції generate_random_number() генерується число, яке розміщується у першому функціональному перетворювачі у комірці, адреса v_last[0] якої була “зарезервована” раніше.

Маючи це значення, можна виконати прохід по решті стовпчиків останнього шару та за допомогою оператора виключного АБО знайти значення w на виході з

функціонального перетворювача послідовно на кожному стовпчику. Це значення w заноситься за відповідною адресою комірки з масиву v_last, що були “зарезервовані” раніше.

На цей момент завершилася генерація нового паролю користувача та заповнення для нього таблиць функціональних перетворювачів. На цій ітерації залишилося зробити усього декілька кроків.

По-перше, перевірити значення налаштування *CHECK_DUPLICATES*, та у випадку, якщо воно встановлено у значення true, то перевірити, що новозгенерований пароль *x* ще не міститься у списку згенерованих паролів для користувача за допомогою виклику функції *unique(x, xs)*. Якщо раптом такий пароль уже існує, то користувач сповіщається про це відповідним повідомленням, а лічильник duplicates_count збільшує своє значення на 1.

Якщо ж цей пароль є унікальним, то він додається до вектора усіх паролів xs, і користувачу виводиться повідомлення про це і про поточну кількість згенерованих паролів (довжину вектора xs).

По-друге, перевіряється умова, що можна продовжувати генерацію паролів за допомогою виклику функції *all_funcs_sufficient()*. Ця умова є достатньою, адже вона перевіряє наявність мінімальної кількості порожніх комірок у кожному з функціональних перетворювачів, що будуть використані для створення “шляху” для чергового паролю на наступній ітерації. Якщо ця кількість не є достатньою, це свідчить про те що функціональні перетворювачі насичені, і генерацію паролів потрібно завершити.

Усі комірки з порожніми значеннями, що залишилися після роботи алгоритму, заповнюються значеннями, згенерованими за допомогою функції *generate_random_number()*.

Користувачу виводиться повідомлення про те, що алгоритм завершив свою роботу. Якщо встановлено налаштування про необхідність перевірки згенерованих

паролів, то це робиться з використанням функції hashes_match(). На екран виводиться кількість згенерованих паролів.

На цьому закінчується виконання програмного застосунку.

4.3. Налаштування роботи програми

Налаштування роботи програми виконується за допомогою встановлення відповідних значень для множини констант, що наведені далі, які знаходяться на початку файлу *main.rs*:

- *FRAGMENTS_BITNESS* – бітова довжина одного фрагмента вхідного та вихідного векторів, відповідає числу *K* в алгоритмі. Такий самий розмір матимуть комірки функціональних перетворювачів, а також усі числа, над якими виконуватимуться операції у процесі роботи алгоритму;

- *FRAGMENTS_COUNT* – задає кількість фрагментів для розбиття ідентифікатора користувача і відповідає числу *M* в алгоритмі. Оскільки один такий фрагмент складається з *FRAGMENTS_BITNESS* біт, то сумарна розрядність вихідного вектору обраховується як *FRAGMENTS_BITNESS*FRAGMENTS_COUNT*;

- *CHECK_DUPLICATES* — задає, чи необхідно реалізовувати механізм перевірки нових паролів користувача, що генеруються, на предмет повторення між собою, для запобігання створення дублікатів у вихідній множині паролів. Беручи до уваги той факт, що вірогідність співпадіння двух створених паролів є дуже низькою, то рекомендується залишати значення цього налаштування у стані *false*, адже це відключає додаткові перевірки по ходу роботи програми та зменшує час її виконання.

- *CHECK_KEYS* — задає, чи необхідно реалізовувати механізм перевірки нових паролів користувача, що генеруються, на правильність, тобто що пропускання кожного з паролів через створене булеве функціональне перетворення

дає на виході одне і те саме значення ідентифікатора користувача, для якого вони генеруються. Оскільки, згідно логіки роботи алгоритму, паролі генеруються саме таким чином, аби давати правильний результат на виході булевого функціонального перетворювача, то рекомендуються залишати значення цього налаштування у стані *false*, адже це відключає додаткові перевірки по ходу роботи програми та зменшує час її виконання;

- *COMMENT* — задає, чи потрібно детально коментувати виконання алгоритму по ходу роботи програми та виводити на екран проміжні значення. Оскільки для великих значень розмірності параметрів роботи програми вона буде генерувати велику кількість таких повідомлень, то рекомендується переключати значення цього параметру у стан *true* лише для тестових запусків та при малих розмірностях чисел і архітектури, а інакше залишати його у стані *false*.

Глобальні налаштування програмного застосунку, які обраховується виходячи з параметрів

На основі описаних вище параметрів налаштування роботи програми будуть підраховані це декілька глобальних значень, які використовуватимуться по всьому ході роботи програми та які варто явно описати:

- *NUMBERS_COUNT* – визначає розмір (довжину) таблиць функціональних перетворювачів, тобто кількість комірок у кожному з них. Обраховується як $2^{\text{FRAGMENTS\_BITNESS}}$;

- *MAX_NUMBER* – визначає максимальне коректне значення для чисел, які зберігаються у комірках та над якими проводяться операції по ходу роботи алгоритму. Використовується як верхня границя при генерації чисел з використанням псевдовипадкового генератора. Обраховується як (*NUMBERS_COUNT — 1)*;

- *EMPTY_NUMBER* — визначає числове значення, яке використовується у програмному застосунку для позначення порожніх комірок або неможливих станів.

Встановлюється рівним *NUMBERS_COUNT*, оскільки усі коректні числа, що будуть використовуватися, будуть точно меншими за це значення;

- *YS_COUNT* — визначає довжину масиву, що зберігає фрагменти вихідного вектору (ідентифікатору користувача). Встановлюється рівним *FRAGMENTS_COUNT*, оскільки вона має таке числове значення за визначенням;
- *XS_COUNT* — визначає довжину масиву, що зберігає фрагменти вхідного вектору (паролю користувача). Встановлюється рівним (*2*FRAGMENTS_COUNT-1)*, оскільки вона має таке числове значення згідно архітектури булевого функціонального перетворювача, що використовується;
- *FUNCS_COUNT* — визначає довжину масиву, що зберігає функціональні перетворювачі, які використовуються у роботі алгоритму для заданих параметрів архітектури. Встановлюється рівним *XS_COUNT*, оскільки вона має саме таке числове значення згідно архітектури булевого функціонального перетворювача, адже кількість фрагментів вхідного вектору співпадає з кількістю необхідних на першому шарі функціональних перетворювачів.

Інструкція користувача

Управління роботою програмного застосунку відбувається напряму через внесення змін до вихідного коду з наступною його перекомпіляцією та генеруванням нового бінарнику для запуску.

Тому першим кроком є відкриття вихідного файлу *main.rs*, що знаходиться у директорії *src* проекту, будь-яким файловим редактором.

Для встановлення параметрів роботи програмного застосунку використовується декілька констант на самому початку файлу. До них належать:

1. FRAGMENTS_BITNESS
2. FRAGMENTS_COUNT
3. CHECK_DUPLICATES
4. CHECK_KEYS

5. COMMENT

Перша та друга константи встановлюють бітову довжину та кількість фрагментів вихідного вектору, відповідно.

Третя, четверта та п’ята змінні встановлюють допоміжні параметри роботи програмного застосунку, а саме необхідність виконання перевірки згенерованих паролів на дублікати, необхідність перевірки коректності згенерованих паролів, а також чи потрібно виводити детальну інформацію по ходу роботи програми про роботу алгоритму.

Зробивши необхідні зміни у файлі main.rs, необхідно зберегти їх та відкрити термінал у директорії проекту. Звідти потрібно запустити команду *cargo run --release*, яка згенерує новий бінарник з максимальним рівнем оптимізації. Якщо виникли які-небудь синтаксичні помилки, компілятор повідомить про це. У такому разі треба повернутися до файлу *main.rs* та занести необхідні правки, після чого повторити компіляцію.

У разі успішної компіляції буде створено оптимізований та готовий до запуску бінарник за шляхом *./target/release/main*.

Останнім кроком є запуск цього бінарнику на виконання. Від користувача не потребується ніяка взаємодія з програмою по ходу її виконання. На екран буде виводитися прогрес генерації паролів та їх остаточна кількість, а також допоміжні коментарі, якщо встановлено налаштування *COMMENT*.

На цьому завершується робота програмного застосунку.

4.4. Розробка функціональної та принципової схеми процесора ідентифікації

Виконання ідентифікації користувача у системі — процес, що виконується постійно, багато разів та, нерідко, для багатьох користувачів одночасно. Оскільки етап проходження ідентифікації користувачем у системі має займати як можна

менше часу, а також зважаючи на той факт, що запропонований метод ідентифікації включає в себе певну послідовність кроків, що займають деякий час, то очевидно, що важливо не просто реалізувати механізм ідентифікації, але й зробити можливість його виконання з належною швидкодією.

Запропонований метод ідентифікації заснований на булевому функціональному перетворювачі. Однією з причин вибору саме булевих функціональних перетворювачів у якості алгебраїчного базису обумовлювалося не в останню чергу можливістю її ефективної апаратної реалізації.

Дійсно, описана у другому розділі архітектура булевого функціонального перетворювача передбачає використання лише двох базових складових — функціонального перетворювача у вигляді табличного блоку пам’яті, який видає значення, що зберігається за заданою адресою, а також блоку булевої операції виключного АБО.

Обидва блоки є давно вивченими та простими з точки зору апаратної реалізації, а тому апаратна реалізація описаного у другому розділі алгоритму ідентифікації буде мати значно більшу швидкодію, аніж її програмний аналог.

Завдання реалізації розповсюджених алгоритмів шифрування на спеціальних апаратних вузлах, що отримали назву криптографічних сопроцесорів, є давно відомим та вивченим. До уже реалізованих апаратно у таких криптографічних сопроцесорах алгоритмів можна віднести алгоритм шифрування з Rijndael, алгоритм несиметричного шифрування Ель-Гамаля, алгоритми створення та перевірки цифрових підписів з використанням DSA, та не в останню чергу самі булеві функціональні перетворення.

Для того аби розробити функціональну схему такого криптографічного сопроцесора першим кроком є прийняття рішення про доцільність використання тих чи інших операційних вузлів, які якнайкраще підходять для виконання конкретного алгоритму апаратно. Наступним кроком є отримання переліку вхідних

та вихідних сигналів до цих вузлів, за допомогою яких відбувається керування їх роботи. Отримавши ці сигнали, необхідно описати їх взаємозв’язки та загальну топологію системи.

При апаратній реалізації алгоритму одним з головних питань є практичний аспект такої реалізації. Для того аби його можливо було використовувати замість його програмного аналогу, він має мати низьку собівартість, для того аби загальна ціна його використання виправдала затрати на неї.

Тому очевидним рішенням є використання криптографічного сопроцесору, який має гарну інтегрованість та здатен забезпечити високу швидкодію і, що не мало важливо, випускається серійно. Такі сопроцесори випускаються масово починаючи з кінця двадцятого століття, а з початку нульових з’явилися сопроцесори, що реалізують одразу декілька алгоритмів — потрібно лише обрати необхідний.

Поставленим вимогам відповідає криптографічний сопроцесор, що був розроблений та випущений у 2009 році — модель 6500, який містить у собі апаратну реалізацію декількох криптографічних алгоритмів та здатен оперувати числами з бітовою довжиною, рівною 2048. Саме він був обраний у якості криптографічного сопроцесора для реалізації описаного алгоритму ідентифікації.

Далі буде розглянуто апаратну складову цього криптографічного сопроцесору.

До його складової входить, по-перше, набір шістнадцяти регістрів для зберігання даних, кожен з яких має бітову довжину 2^{11}. Для того аби знати реальну довжину даних, що записується у регістри, кожному з них відповідає додаткових регістр, який має бітову довжину 11, який автоматично зберігає реальну довжину числа, що записується у основний регістр. Аналіз показав, що пам’яті, яка є доступною завдяки використанню регістрів, достатньо для реалізації алгоритму, а тому додатковий зовнішній блок пам’яті не використовується. По-друге, до складу

сопроцесора входить АЛП з бітовою довжиною операндів, рівною 2^{10}. По-третє, він містить спеціальний генератор псевдовипадкових чисел, що має швидку апаратну реалізацію на основі внутрішніх тактових генераторів, при цьому швидкість генерування псевдовипадкового сигналу досягає трьох мегабіт за секунду.

Для управління роботою усіх вузлів використовується окремий блок управління, що приймає набір інструкції пачками та самостійно відправляє їх на конвейер для виконання. Для доступу до окремих вузлів сопроцесор має шість шин, бітова довжина кожної з яких є рівною 2^{10}.

Оскільки цей криптографічний сопроцесор відноситься до класу периферійних пристроїв, для яких обмін даними є на декілька порядків повільнішим за швидкість його роботи, то він підключається до основної комп’ютерної системи за допомогою інтерфейсу PCI

Можна сказати, що система, що проектується, складається з безпосередньо самого криптографічного сопроцесора, а також спеціального вузла для його підключення до шини PCI. Цей вузол по факту є адаптером між шинами PCI та ATA.

В основному такі адаптери використовуються для можливості підключення оптичних та магнітних накопичувачів, що стандартно використовують для роботи інтерфейс ATA, до швидкісної шини PCI.

Потрібно використати альтернативну до стандартної реалізації такого адаптеру між PCI та ATA, оскільки контролери дискових накопичувачів мають значно нижчу швидкість роботи, аніж запропонований криптографічний сопроцесор. Через це необхідно зробити так, аби сопроцесор мав більший пріоритет і не сповільнювався адаптером, що використовується.

До функціонального складу адаптеру, окрім розведення контактів між двома 4-байтними шинами з обох боків, входять також допоміжні регістри та часові таймери, які виконують синхронізацію між двома інтерфейсами, а регістри в свою

чергу зберігають інформацію, що надсилається в обидва боки, доки вона не була прийнята протилежною стороною.

Через особливість роботи інтерфейсу PCI, а саме через те, що для передачі даних використовуються ті самі лінії, що і для передачі адрес, виникає необхідність для додання спеціального регістру на 4 байти для зберігання поточної адреси, яка приходить з центрального процесору. При цьому старша шістнадцяти-бітова частина адреси порівнюється з поточною при виконанні транзакції, і у разі їх співпадіння, молодші дванадцять розрядів передаються до інтерфейсу ATA.

Описаний адаптер між PCI та ATA використовує два окремих сигнали для синхронізації для кожного з інтерфейсів окремо. При цьому для інтерфейсу PCI використовується сигнал clock зі слоту розширення, у який вставляється адаптер. У якості сигналу синхронізації для інтерфейсу ATA використовується зовнішній сигнал зі схеми для генерації імпульсів, причому цей самий сигнал використовується для роботи самого криптографічного сопроцесору. Сам сопроцесор містить внутрішній кварцевий генератор. Різниця між фазами цих двох сигналів використовується у якості вхідного сигналу для генерації псевдовипадкових чисел.

Далі буде описано склад принципової схеми сопроцесора.

Як вже зазначалося раніше, розроблена схема складається з двох головних елементів — безпосередньо сам сопроцесор (що позначений DD2 на схемі), а також адаптер сполучення сопроцесору з інтерфейсом PCI. Цей інтерфейс використовується для того, аби реалізувати фізичний та логічний канал зв’язку між інтерфейсом PCI і шиною ATA, через яку виконується підключення сопроцесору у якості периферійного пристрою до комп’ютера. У якості мікросхеми для цього адаптеру була вибрана мікросхема XR16M654 (позначена на схемі як DD1). Вона представляє собою стандартний блок, що має 186 лапок-виходів і живиться двома рівнями напруг — три та п’ять вольт. На схемі використовуються конденсатори

електролітичного типу для того аби згладити стрибки напруги на вході живлення — по одному на кожну лінію напруги (позначені як C1 та C2). З метою запобігання пошкодженню схеми високочастотними шумами використовується високочастотний фільтр на основі конденсаторів керамічного типу (позначені як C з індексами від 4 до 11).

Ця мікросхема має стандартний інтерфейс PCI, тому може бути підключена до будь-якого із вільних слотів материнської плати. Лінійний випрямовувач сигналу потребує додаткового резистора з опором 10^4 Ом (позначений через R1 на схемі, точка B), через який він підключається до п’ятивольтової лінії, що слугуватиме у якості рівня логічної одиниці.

Як вже зазначалося раніше, для синхронізації використовуються сигнали *clock*, який іде по слоту розширення південного мосту і забезпечує синхронізацію для інтерфейсу PCI, а також за допомогою зовнішньої мікросхеми, що генерує імпульси, яка використовується для синхронізації самого адаптеру по шині ATA. Обидва інтерфейси ділять спільний сигнал *reset*, що приходить від південного мосту до інтерфейсу ATA, а потів і до самого сопроцесора.

Вхідні та вихідні контакти криптографічного сопроцесора з’єднуються з одноіменними контактами інтерфейсу ATA.

До складу схеми входить кварцевий резонатор (позначений DD3), який здатен видавати стабільну частоту у 10^8 герц, і використовується для забезпечення синхронізації блоку адаптера і сопроцесора ззовні. На цій частоті працює сам сопроцесор і складові елементи блоку адаптера. Оскільки зовнішньою частотою для адаптера PCI-ATA є частота, що є утричі меншою, то для її генерації використовується дільник частоти, що входить до складу південного мосту. Інверсний контакт вибору кристала виведений напряму до рівня логічного нуля у системі.

Для досягнення бажаного перепаду імпульсів з виходу мікросхеми використовується (точка А) додатковий резистор такого ж опору 10^5 Ом до п’ятивольтового рівня напруги.

При підключенні будь-якої периферійної апаратури у розширювальний слоти материнської плати необхідно перевірити, що вони не перевантажать систему, споживаючи більшу потужність, аніж система вона здатна забезпечити.

Оскільки схема, що проектується, буде підключатися до інтерфейсу PCI системи, то необхідно перевірити, що її споживча потужність не перевищуватиме 22 Ватт, що є лімітом потужності згідно документації інтерфейсу PCI. Тому важливо провести аналіз компонентів, що входять до її складу, та перевірити їх сумарну споживчу потужність.

Для схеми, що проектується, основними компонентами, що споживають потужність, є мікросхеми. Таблиця 4.1 містить інформацію щодо окремих споживчих потужностей кожного вузла системи криптографічного сопроцесору, що розробляється.

Таблиця 4.1

Потужність, що споживається окремими вузлами системи

Вузол системи	Умовне позначення на схемі	Потужність, що споживається вузлом, Ватт
Адаптер PCI-ATA	DD1	8.2
Криптопроцесор	DD2	6.3
Кварцевий резонатор	DD3	0,4

Згідно даних, наведених у таблиці, видно, що загальна сума потужностей складає 14.9 Ватт, що є меншим за ліміт інтерфейсу PCI, а отже розроблена схема криптографічного сопроцесора відповідає може бути безпечно підключена до нього.

Висновки до розділу 4

По проведеній у четвертому розділі роботі по створенню програмного застосунку та тестуванню описаного у другому розділі алгоритму за допомогою цього застосунку можна зробити такі висновки:

1. Було реалізовано програмний застосунок на мові програмування Rust, що виконує тестування запропонованого у другому розділі алгоритму побудови булевого функціонального перетворювача. Використання мови програмування Rust для реалізації програмного застосунку дало можливість отримати високошвидкісний та оптимізований бінарник з відносно невеликими вимогами до пам’яті при написанні високорівневого програмного коду.

2. Налаштування параметрів роботи програмного застосунку виконується безпосередньо у файлі програмного коду для максимізації швидкості виконання застосунку завдяки виконанню більшої кількості оптимізацій на етапі компіляції.

3. Створений програмний застосунок дає змогу оцінити кількість паролів користувача, яку генерує для заданих параметрів архітектури булевого функціонального перетворювача запропонований у другому розділі метод. Виявилося, що запропонований метод здатен згенерувати велику кількість паролів користувача з достатньою швидкістю при використанні стандартних довжин криптографічних ключів.

4. Окрім реалізації самого алгоритму побудови булевого функціонального перетворювача, було розроблено ряд додаткових механізмів, що дозволяють перевіряти правильність роботи алгоритму та виводити на екран детальні коментарі про хід роботи алгоритму.

ВИСНОВКИ

Під час виконання магістерської роботи було проведено ретельний огляд розповсюджених методів ідентифікації віддалених користувачів, що існують на сьогоднішній день, та проаналізовано їх переваги і недоліки з урахуванням темпів підвищення інформаційної інтеграції.

На основі проведеного аналізу було показано, що найбільш ефективним з точки зору криптографічного захисту є використання заснованій на концепції “нульових знань” криптографічно-строгої ідентифікації, яка передбачає наявність лише одного власнику секретних даних.

Для можливості ефективної та швидкої апаратної реалізації було прийнято рішення використовувати булеві функціональні перетворювачі у якості алгебраїчного базису хеш-перетворення.

Був описаний новий метод побудови хеш-перетворень на основі булевих функціональних перетворювачів, що відповідає вимогам, які висуваються концепцією “нульових знань”.

Для довизначення отриманого у результаті виконання описаного методу булевого функціонального перетворювача для наборів, на яких він залишився невизначеним після виконання методу, було запропоновано новий спосіб довизначення його функціональних перетворювачів так, аби отримане хеш-перетворювання задовольняло необхідному для високого рівня криптографічного захисту строгому лавинному ефекту.

Було створено набір програмних застосунків для тестування та перевірки роботи запропонованого методу. Тестування показало, що запропонований метод здатен забезпечити генерацію великої кількості паролів та має високу швидкість роботи при досягненні сучасного рівня криптозахисту, і може бути використаний на практиці для реалізації механізму ідентифікації у системах багатьох користувачів.

СПИСОК ЛІТЕРАТУРИ

1. Широчин В.П., Мухин В.Е., Кулик А.В. Вопросы проектирования средств защиты информации в компьютерных системах и сетях. К.: 2000. – 111 с.

2. Bardis N., Doukas N. Markovskyi O. A Method for strict remote user authentication using non-revesible Galois field transformations // Proceeding of IEEE Symposium on Computers and Communications. ISCC-2017. 3-6 July 2017. Heraclion, Crete, Greece. – P.243-249.

3. Гук М.Ю. Аппаратные интерфейсы IBM PC. Энциклопедия. – СПб.: Питер. – 2002. – 495 с.

4. Иванов М.А., Криптографические методы защиты информации в компьютерных системах и сетях. М.: ”Кудиз-образ”, 2001. – 368 с.

5. Захарченко Н.А., Топалова К.Н. Использование булевых преобразований для быстрой идентификации абонентов на основе концепции нулевых знаний. // Матеріали XII Міжнародної науково-технічної конференції ”Системний аналіз та інформаційні технології”. – К.:НТУУ ”КПІ”. – 2010. – С.441.

6. Захариудакис Лефтерис. Метод быстрой аутентификации удаленных пользователей на основе концепции “нулевых знаний” /Наукові записки Українського науково-дослідного інституту зв’язку. 2017. – № 1 (45). – С.109-117.

7. Мухін В.Є. Метод ідентифікації віддалених абонентів на основі концепції “нульових знань” / В.Є. Мухін, Лефтеріс Захаріудакіс, Ю.Н. Герасименко, М.С. Козерацький // Телекомунікаційні та інформаційні технології, 2017 – №1. – С.50-57.

8. Самофалов К.Г., Тубольцев А.А., Ияд Мохд Маджид Ахмад Шахрури. Повышение эффективности идентификации удаленных абонентов многопользовательских систем // Проблеми інформатизації та управління. Збірник наукових праць: Випуск 3(14). – К.,НАУ. – 2008. – С.129-136.

9. Самофалов К.Г., Марковский А.П., Гаваагийн Улзисайхан, Бардис Н., Метод получения булевых балансных SAC-функций для систем защиты информации. // Вісник Національного технічного університету України ”КПІ”. Інформатика, управління та обчислювальна техніка. – 1998. – № 31. – С.131-140.

10. Budaghyan J. Construction and Analysis of cryptographic functions. New York,:NY:Springer-Verlag. – 2014. – 200 p.

11. Burrows M., Abadi M., Needham R. A logic for authentication // Proceeding of the 12-th ASM Symposium on Operating Systems Principles. N.Y. – 1989. – P.39-48.

12. Feige U., Fiat A., Shamir A. Zero knowledge proofs of identity // Journal of Cryptology. – Vol. 1. – № 2. – 1988. – P.77-94.

13. Knudsen L.N. The block cipher companion / L.R. Knudsen, M.J.B. Robashaw // Berlin:Springer-Verlag. – 2011.

14. Kittichokenai K. Secret Key-based Identification and Authentication with a Privacy Constraint / K. Kittichokenai, G. Caire // IEEE Trans. Inf. Theory. – Vol. 62. – 2016. – № 11. – P. 6189-6203.

15. Марковский А.П., Эль-Хами И., Рябуха Л.Р. Метод одержання булевих балансних функцій, які задовольняють критерію чіткого лавинного ефекту // Наукові Вісті НТУУ “КПІ“. – 2001. – № 2. – с.31-40.

16. Kurosawa K., Satoh T. Design of SAC/PC(l) of Order k Boolean Functions and Three Other Cryptographic Criteria // Advanced in Cryptology – Eurocrypto’97 Proceeding, Lecture Notes in Computer Science 1233 – 1997. – P.433-449.

17. Самофалов К.Г., Марковский А.П. Комбинаторный подход к получению булевых функций, обладающих строгим лавинным эффектом // Электронное моделирование. – 2004. – Том. 26. – № 3. – с.27-40.

18. Yang Y., Wang S., Bao F., Wang J., Deng R. H. New efficient user identification and key distribution scheme providing enhanced security // Computers & Security. – Vol. 23. – 2004. – P. 697-704.

19. Todorov D. Mechanics of User Identification and Authentication: Fundamentals of Identity Management. – 2007. – P.760.

20. Behera P., Gangopadhyay S. An improved hybrid genetic algorithm to construct balanced Boolean function with optimal cryptographic properties. // Evolutionary Intelligence, Springer. – 2021. – P.1-15.

21. Марковський О.П. Метод строгої ідентифікації віддалених абонентів на основі стандартизованих шифроблоків та хеш-перетворень / О.П. Марковський, Захаріудакіс Лефтеріс., М.Ф. Федотов // Вісник Національного технічного університету України “КПІ” Інформатика, управління та обчислювальна техніка. К.: ТОО „ВЕК+”. 2016. – № 64. – С. 161-165.

22. Марковский А.П., Зюзя А.А., Шерстюк В.Д. Получение булевых преобразований специальных классов для построения эффективных алгоритмов защиты информации // Вісник Національного технічного університету України ”КПІ”. Інформатика, управління та обчислювальна техніка. – К.: ”ВЕК++” – 2008. – № 49. – С.7-13.

23. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си. – М.:”Триумф”. – 2002. – 816 с.

24. Chaum D., Van Heijst E., Pfitzmann B. Cryptographicall strong undeniable signatures, unconditionally secure for signer, Advances in Cryptology - Crypto ’91, v. 576 of Lecture Notes in Computer Science. – 1992. – P. 470-484.

25. Fiat A., Shamir A. How to prove yourself: practical solutions to identification and signature problems // Proceeding of Workshop ”Crypto’86”. – Santa Barbara, USA, August 11-15, 1986. – Berlin: Springer-Verlag. – LNCS-263. – 1986. – P.186-194.

26. Goldreich O. On concurrent identification protocols // Proceeding of Workshop ”Eurocrypt’84”. – Paris, France, April 9-11, 1984. – Berlin: Springer-Verlag. – LNCS-209. – 1984. – P.387-398.

27. Kurosawa K., Yoshida T. Strongly universal hashing and identification codes via channels / K. Kurosawa, T. Yoshida // IEEE Trans. Information theory. – Vol. 45. – № 6. – 1999. – P.2091-2095.

28. Wang W. On the Relation Between Identifiability, differential Privacy and Mutual Informational Privacy / Wang W., Ying I., Zhang J. // IEEE Trans. Inf. Theory. – Vol. 62. – 2016. – № 9. – P. 5018-5029.

29. J. von Zur Gathen Modern Computer Algebra, 3rd ed / J. von Zur Gathen, J. Gerhard // New York, NY, Cambridge Univ. Press. – 2013.

30. Cusick T. Cryptographic Boolean Functions and Applications, 2nd ed / T. Cusick, P. Stanica // Academic Press. – 2017. – P.288.

31. Boyarshin I. Method of hash transformations construction for strict user identification / Igor Boyarshin, Oleksandr Markovskyi // International Conference ICSFTI2019 (Kyiv, May 14−15, 2019). Kyiv, 2019. – P. 41-46.

32. Rusanova O. Energy-aware task scheduling algorithm for mobile computing / Olga Rusanova, Igor Boyarshin, Anna Doroshenko // International Conference ICSFTI2020 (Kyiv, May 13, June 15, 20120). Kyiv, 2020. – P. 107-113.

33. Boyarshin I. Request balancing method for increasing their processing efficiency with information duplication in a distributed data storage system / I. Boyarshin, A. Doroshenko, P. Rehida // Technical sciences and technologies. – 2021. – № 2 (26).

34. Crama Y. Boolean Models and Methods in Mathematics, Computer Science, and Engineering / Yves Crama, Peter L. Hammer // Cambridge University Press. – 2010. – P. 759.

35. Burnett L. Simpler methods for generating better Boolean functions with good cryptographic properties / Linda Burnett, William Millan, Ed Dawson, Andrew Clark // Australasian Journal of Combinatorics. – Vol. 29. – 2004. P. 231-247.

36. Tokareva N. Bent Functions: Results and Applications to Cryptography / Natalia Tokareva // Academic Press. – 2015. – P. 220.

37. Asthana R. Generation of Boolean functions using Genetic Algorithm for cryptographic applications / R. Asthana, N. Verma and R. Ratan // IEEE International Advance Computing Conference (IACC). – 2014. – P. 1361-1366.

38. Fuller J. Analysis of affine equivalent boolean functions for cryptography / Joanne Fuller // PhD thesis, Queensland University of Technology. – 2003.

39. Boros E. Extensions of Partially Defined Boolean Functions with Missing Data / Endre Boros, Toshihide Ibaraki, Kazuhisa Makino // Rutgers University. – 1996.

40. Singh M. Choosing Best Hashing Strategies and Hash Functions / M. Singh, D. Garg // IEEE International Advance Computing Conference. – 2009. – P. 50-55.

41. Tzong-Sun Wu Efficient user identification scheme with key distribution preserving anonymity for distributed computer networks / Tzong-Sun Wu, Chien-Lung Hsu // Computers & Security. – Vol. 23 (2). – 2004. P. 120-125.

42. Bakhtiari S. Cryptographic Hash Functions: A Survey / Sedigheh Bakhtiari, Reihaneh Safavi-Naini, Josef Pieprzyk // Centre for Computer Security Research, Department of Computer Science, University of Wollongong, Australia. – 1995.

43. Todorov D. Mechanics of User Identification and Authentication: Fundamentals of Identity Management / Dobromir Todorov // CRC Press. – 2007. – P. 760.

44. Girault M. On the length of cryptographic hash-values used in identification schemes / Girault M., Stern J. // Advances in Cryptology – CRYPTO ’94. Berlin: Springer-Heidelberg. – 1994.

45. Zhang Jiajing Two-Stage User Identification Based on User Topology Dynamic Community Clustering / Zhang Jiajing, Zhenhua Yuan, Neng Xu, Jinlan Chen, Wang Juxiang // Complexity. – 2021. – P. 1-10.

46. Huang Yuhua A Lightweight Hash Function Based on Dual Pseudo-Random Transformation / Huang Yuhua, Li Shen, Sun Wanlin, Dai Xuejun, Zhu Wei // HVH. – 2021. – P. 492-505.

47. Kishore Neha Enveloped Inverted Tree Recursive Hashing: An Efficient Transformation for Parallel Hashing / Kishore Neha, Raina Priya // Advances in Communication and Computational Technology, Select Proceedings of ICACCT 2019. – 2020. – P. 481-499.

Додатки

Додаток А

Лістинг програми

main.rs

```
use rand::Rng;

const FRAGMENTS_BITNESS : usize = 18; // K
const FRAGMENTS_COUNT   : usize = 9; // M
const NUMBERS_COUNT     : usize = 1 << FRAGMENTS_BITNESS;
const MAX_NUMBER        : usize = NUMBERS_COUNT - 1;
const EMPTY_NUMBER      : usize = NUMBERS_COUNT;
const YS_COUNT          : usize = FRAGMENTS_COUNT;
const XS_COUNT          : usize = 2 * FRAGMENTS_COUNT - 1;
const FUNCS_COUNT       : usize = XS_COUNT;
const CHECK_DUPLICATES  : bool = false;
const CHECK_KEYS        : bool = false;
const COMMENT           : bool = false;
// ----------------------------------------------------------------------------
#[derive(PartialEq, Clone, Copy)]
struct Number {
    value: usize,
}

impl Number {
    fn empty() -> Number {
        Number { value: EMPTY_NUMBER }
    }

    fn new(value: usize) -> Number {
        Number { value }
    }

    fn is_empty(&self) -> bool {
        self.value == EMPTY_NUMBER
    }

    fn xor(&self, other: &Number) -> Number {
        Number::new(self.value ^ other.value)
    }
}

impl std::fmt::Display for Number {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        if self.is_empty() {
            write!(f, "*")
        } else {
            write!(f, "{}", self.value)
        }
    }
}
// ----------------------------------------------------------------------------
type FragmentsOfX = [Number; XS_COUNT];
type FragmentsOfY = [Number; YS_COUNT];
```

```
#[allow(dead_code)]
fn equal(a: &FragmentsOfX, b: &FragmentsOfX) -> bool {
    a.iter().eq(b.iter())
}

fn not_equal(a: &FragmentsOfX, b: &FragmentsOfX) -> bool {
    a.iter().ne(b.iter())
}

fn not_equal_array_y(a: &FragmentsOfY, b: &FragmentsOfY) -> bool {
    a.iter().ne(b.iter())
}

fn unique(t: &FragmentsOfX, ts: &Vec<FragmentsOfX>) -> bool {
    ts.iter().all(|el| not_equal(el, t))
}

fn x_to_y(x: FragmentsOfX) -> FragmentsOfY {
    let mut y: FragmentsOfY = [Number::empty(); YS_COUNT];
    for i in 0..(YS_COUNT) {
        y[i] = x[i];
    }
    y
}

#[allow(dead_code)]
fn print_array_x(x: &FragmentsOfX) {
    print!("(");
    for i in 0..(XS_COUNT - 1) {
        print!("{},", x[i]);
    }
    print!("{}", x[XS_COUNT - 1]);
    print!(")");
}

fn print_array_y(y: &FragmentsOfY) {
    print!("(");
    for i in 0..(YS_COUNT - 1) {
        print!("{},", y[i]);
    }
    print!("{}", y[YS_COUNT - 1]);
    print!(")");
}
// ------------------------------------------------------------------------
// Range: [low; high]
fn generate_random_uniform_int(random: &mut rand::rngs::ThreadRng, low: usize, high:
usize) -> usize {
    random.gen_range(low..=high) as usize
}

fn generate_random_number(random: &mut rand::rngs::ThreadRng) -> Number {
    Number::new(generate_random_uniform_int(random, 0, MAX_NUMBER))
}

fn generate_random_array_y(random: &mut rand::rngs::ThreadRng) -> FragmentsOfY {
    let mut y: FragmentsOfY = [Number::empty(); YS_COUNT];
```

```
    for i in 0..YS_COUNT {
        y[i] = Number::new(generate_random_uniform_int(random, 0, MAX_NUMBER));
    }
    y
}
// ------------------------------------------------------------------------
struct Func {
    map: Vec<Number>,
    cache: Vec<Vec<Number>>,
    empty_count: usize,
}

impl Func {
    fn new() -> Func {
        let mut map   = Vec::new();
        let mut cache = Vec::new();
        map.resize(NUMBERS_COUNT, Number::empty());
        cache.resize(NUMBERS_COUNT, Vec::new());
        Func {
            map,
            cache,
            empty_count: NUMBERS_COUNT,
        }
    }

    fn set(&mut self, arg: Number, value: Number) {
        if self.map[arg.value].is_empty() != value.is_empty() {
            if value.is_empty() {
                self.empty_count += 1;
            } else {
                self.empty_count -= 1;
            }
        }

        if value.is_empty() {
            let key = self.map[arg.value];
            let items = &mut self.cache[key.value];
            items.retain(|&x| x.value != arg.value);
            // let index = items.iter().position(|&v| v.value == arg.value);
        } else {
            self.cache[value.value].push(arg);
        }

        self.map[arg.value] = value;
    }

    fn at(&self, arg: usize) -> Number {
        self.map[arg]
    }

    #[allow(dead_code)]
    fn empty_at(&self, arg: Number) -> bool {
        self.map[arg.value].is_empty()
    }

    fn get_empty_count(&self) -> usize {
        self.empty_count
```

```
    }
}

// type Funcs = [Func; FUNCS_COUNT];

#[allow(dead_code)]
fn print_funcs(funcs: &Vec<Func>) {
    println!("Funcs:");
    for i in 0..NUMBERS_COUNT {
        for f in funcs {
            print!("{} -> {}\t\t", i, f.at(i));
        }
        println!();
    }
}

fn run_through(funcs: &Vec<Func>, start_x: &FragmentsOfX) -> FragmentsOfY {
    let mut x = start_x.clone();
    for width in ((YS_COUNT + 1)..=XS_COUNT).rev() {
        for j in 0..width {
            x[j] = funcs[j].at(x[j].value);
        }
        for j in 0..(width - 1) {
            x[j] = x[j].xor(&x[j + 1]);
        }
    }
    for j in 0..FRAGMENTS_COUNT {
        x[j] = x[j].xor(&start_x[j]);
    }
    x_to_y(x)
}

fn hashed_match(funcs: &Vec<Func>, xs: &Vec<FragmentsOfX>, y: &FragmentsOfY) -> bool {
    for x in xs {
        if not_equal_array_y(&run_through(&funcs, x), y) {
            return false;
        }
    }
    true
}

fn pick_random_empty(random: &mut rand::rngs::ThreadRng, func: &Func, forbidden_index:
usize) -> Number {
    if func.get_empty_count() == 0 { return Number::empty(); }
    let starting_index = generate_random_uniform_int(random, 0, NUMBERS_COUNT - 1);
    let mut k = starting_index;
    loop {
        if k == NUMBERS_COUNT { k = 0; }
        if k == forbidden_index { k += 1; continue; }
        if func.at(k).is_empty() { return Number::new(k); }
        k += 1;
    }
}

fn pick_random_not_empty(random: &mut rand::rngs::ThreadRng, func: &Func) -> Number {
    if func.get_empty_count() == NUMBERS_COUNT { return Number::empty(); }
    let starting_index = generate_random_uniform_int(random, 0, NUMBERS_COUNT - 1);
```

```
    let mut k = starting_index;
    loop {
        if k == NUMBERS_COUNT { k = 0; }
        if !func.at(k).is_empty() { return Number::new(k); }
        k += 1;
    }
}

fn pick_random_target(random: &mut rand::rngs::ThreadRng, func: &Func, target: usize) ->
Number {
    let indices = &func.cache[target];
    if indices.is_empty() {
        Number::empty()
    } else {
        let index = generate_random_uniform_int(random, 0, indices.len() - 1);
        indices[index]
    }
}
// ------------------------------------------------------------------------
fn main() {
    // let seed = 314;
    // let mut random = ChaChaRng::seed_from_u64(seed);
    let mut random = rand::thread_rng();
    println!("--------------------BEGIN----------------------");
    println!();
    println!("Running for FRAGMENTS_BITNESS={} and FRAGMENTS_COUNT={}", FRAGMENTS_BITNESS,
FRAGMENTS_COUNT);

    let mut funcs = Vec::with_capacity(FUNCS_COUNT);
    for _ in 0..FUNCS_COUNT {
        funcs.push(Func::new());
    }

    let y = generate_random_array_y(&mut random);
    if COMMENT {
        println!(":> Generated Y = ");
        print_array_y(&y);
        println!();
    }
    let mut xs = Vec::new();

    let all_funcs_sufficient = |funcs: &Vec<Func>| {
        funcs.iter().all(|f| f.get_empty_count() >= FRAGMENTS_COUNT - 1)
    };

    let mut duplicates_count = 0;
    loop {
        if COMMENT { println!(":> Funcs sufficient"); }

        // Step 2
        let mut i = FRAGMENTS_COUNT - 1; // 1-based indexing

        // Step 3
        let mut v_last = [Number::empty(); FRAGMENTS_COUNT + 1];
        for j in 0..v_last.len() {
            v_last[j] = pick_random_empty(&mut random, &funcs[j], EMPTY_NUMBER);
            assert!(!v_last[j].is_empty(), "No empty elements in the last row");
```

```
        }

        let mut old_v = Vec::with_capacity(v_last.len()); // i
        for j in 0..v_last.len() { old_v.push(v_last[j]); } // initial values
        while i >= 2 { // Step 4, 7
            i -= 1; // Step 4
            // Step 5
            let mut v0 = pick_random_not_empty(&mut random, &funcs[0]);
            if v0.is_empty() {
                v0 = pick_random_empty(&mut random, &funcs[0], v_last[0].value);
                assert!(!v0.is_empty(), "Failed to find empty element");
                funcs[0].set(v0, generate_random_number(&mut random)) // w0
            }

            // Step 6
            let mut new_v = Vec::with_capacity(2 * FRAGMENTS_COUNT - 1); // i-1
            new_v.push(v0);
            for j in 1..=2*FRAGMENTS_COUNT-i-1 {
                let prev_v = new_v[j - 1];
                let prev_w = funcs[j - 1].at(prev_v.value);
                let w = prev_w.xor(&old_v[j - 1]);

                let mut v = pick_random_target(&mut random, &funcs[j], w.value);
                if v.is_empty() { // then pick from empty
                    let forbidden = if j < FRAGMENTS_COUNT + 1 { v_last[j].value } else {
EMPTY_NUMBER };
                    v = pick_random_empty(&mut random, &funcs[j], forbidden);
                    assert!(!v.is_empty(), "No empty element found");
                    funcs[j].set(v, w);
                }
                new_v.push(v);
            }

            old_v = new_v;
        } // Step 7

        // Step 8
        let mut x = [Number::empty(); XS_COUNT];
        for j in 0..XS_COUNT {
            x[j] = old_v[j]; // old_v === new_v now
        }

        // Step 9
        funcs[0].set(v_last[0], generate_random_number(&mut random));

        // Step 10
        for j in 1..FRAGMENTS_COUNT+1 {
            let prev_w = funcs[j - 1].at(v_last[j - 1].value);
            let w = x[j - 1].xor(&y[j - 1]).xor(&prev_w);
            funcs[j].set(v_last[j], w);
        }

        // Step 11 is integrared into Func logic
        if CHECK_DUPLICATES && !unique(&x, &xs) {
            println!("-------- Duplicate found --------");
            duplicates_count += 1;
        } else {
```

```
            xs.push(x);
            print!("--- New X generated ({}) ---\r", xs.len());
        }

        if !all_funcs_sufficient(&funcs) { break; } // Step 12
    }

    println!();
    if COMMENT { println!(":> Done. Funcs insufficient!"); }
    println!("Generated {} vectors", xs.len());
    if CHECK_DUPLICATES { println!("{} duplicates discarded", duplicates_count); }

    if CHECK_KEYS && !hashed_match(&funcs, &xs, &y) {
        println!(">>> ERROR: Hash does not match!");
    }

    println!("\n---------------------END-----------------------");
}
```